Coordinating Interactions: The Event Coordination Notation

Kindler, Ekkart

Publication date:
2014

Document Version
Publisher's PDF, also known as Version of record

Coordinating Interactions: The Event Coordination Notation

Ekkart Kindler
Technical University of Denmark
DTU Compute
DK-2800 Kgs. Lyngby
Denmark
ekki@dtu.dk

May 13, 2014

DTU Compute Technical Report 2014-05

DTU Compute - Department of Applied Mathematics and Computer Science
Technical University of Denmark

Building 324
Richard Petersens Plads
DK-2800 Lyngby, Denmark
Phone +45 4525 3031
Fax +45 4588 1399
compute@compute.dtu.dk
www.compute.dtu.dk

Technical Report: ISSN 1601-2321

Abstract

The purpose of a *domain model* is to concisely capture the concepts of an application's domain, and their relation among each other. Even though the main purpose of domain models is not on implementing the application, major parts of an application can be generated from the application's domain models fully automatically with today's technologies. The focus of today's code generation technologies, however, is mostly on the structural aspects of the domain; the domain's behaviour is often not modelled at all, or implemented manually based on some informal models, or the behaviour is modelled on a much more technical level.

The *Event Coordination Notation* (*ECNO*) allows modelling the behaviour of an application on a high level of abstraction that is closer to the application's domain than to the software realizing it. Still, these models contain all necessary details for actually executing the models and for generating code from them.

In order to be able to model the behaviour of a domain, the ECNO makes the *events* in which the different elements of the domain could engage explicit. The *local behaviour* of an element defines at which time an element can engage or participate in an event. The *global behaviour* of the application results from different elements jointly engaging in such events, which is called an *interaction*. Which events are supposed to be jointly executed and which elements need to join in is defined by so-called *coordination diagrams* of the ECNO. Together, the models for the local and the global behaviour define the overall behaviour of the domain.

In this technical report, we discuss the main idea and philosophy of ECNO and its notation as well as all the subtle details and concepts – and we motivate the decisions made for its design. Moreover, we discuss the prototypical implementation of ECNO, which consists of a modelling environment based on Eclipse and the Eclipse Modeling Framework (EMF) and an execution engine, which fully supports all the concepts and features of ECNO discussed in this technical report. All the examples are based on EMF, but the ECNO Engine can be used with different other platforms or object-oriented code across different platforms, once some adapters are provided.

Though the focus of this technical report is on the general concepts of ECNO, the examples discussed here work for version 0.3.2 of the ECNO Tool and Framework. The ECNO Tool as well as the examples are available from the ECNO Home page: `http://www2.imm.dtu.dk/~ekki/projects/ECNO/`.

Contents

Preface

The ideas discussed in this technical report, the concepts of the Event Coordination Notation (ECNO), and the way how theses concepts play together have developed over a long time. The feeling that there is a need for something else capturing coordination on top of classical object-oriented models dates back so long that it is not even possible to put a date to it.

The first tangible ideas of what we call ECNO now came up and where presented in the context of AMFIBIA [2], when we tried to capture the core concepts and main aspects of business process models independently from a specific modelling notation. At that time, we introduced an ad-hoc notation for capturing the intended behaviour of these models. And the core concepts for coordination of that ad-hoc notation are still the same in ECNO today. From that seed, the concepts of ECNO have gradually evolved over time [49, 38, 30, 33, 32, 36, 37]: by adding and generalizing constructs and stripping away others again, by removing unnecessary restrictions, and by building a tool which integrates with existing model-based technologies.

Actually, the ideas of ECNO are still evolving, which, in particular, applies to the ECNO methodology, which is still in its infancy. But, the ideas of ECNO seem to be in a more stable stage now, so that they are ready for being presented to a wider audience – for evaluation, discussion, and further improvement. I am looking forward to any feedback.

Acknowlegements Over time, many people, colleagues as well as students, have contributed to ECNO by commenting and challenging early publications and by making these ideas work in practice. Here is a list of people who, in one form or the other, have contributed to what ECNO is now:

> Achim Heynen, André Altenau, Björn Axenath, Christiane Klapdohr, David Schmelter, Dennis Goeken, Elmar Köhler, Elżbieta Pielenz, Jesper Jepsen, Lukasz Nowak, Maciej Zarzycki, Patrick Könemann, Peter Pietrzyk, Piotr Borowian, Ranghild van der Straeten, Steve Hostettler, Tigran Tchougourian, Vaidas Karosas, Vladimir Rubin, and Yang Li.

It is almost impossible to attribute specific contributions to specific persons. Therefore, I would like to thank all of them together for their work, their ideas, and many interesting discussions.

Specifically, I would like to mention David Schmelter for the implementation of an execution engine of MoDowA [49], which can be considered an early pre-cursor of ECNO. Moreover, I would like to mention Jesper Jepsen for a first "beyond-Mickey-Mouse" example using (almost) the current version of ECNO for modelling a workflow engine [26], which brings us back to where the ideas of ECNO originally started out: modelling the concepts of business process models; only now the models "are" the workflow engine.

May 2014,
Ekkart Kindler,

Chapter 1

Introduction

Long before the advent of *Model-based Software Engineering* (*MBSE*), and one of its main driving forces, the *Model-driven Architecture* (*MDA*) [43], there was an endeavor to better understand and distill the nature of communication and interaction in concurrent systems – with pioneering work of Petri [46], Hoare [21, 22], Harel [17], and Milner [41] identifying foundational concepts, which are still valid today. And there are many different modelling notations for different kinds of purposes for modelling behaviour of distributed, concurrent or cooperating systems based on their concepts.

With the advent of Model-based Software Engineering, models became more and more attention – with the promise that major parts a software system could be generated from these models. Using technologies like the Eclipse Modeling Framework (EMF) [8] can save a lot of programming, making software development significantly faster and the resulting software more reliable. Most of the code generation, however, concerns the structural parts of the software and not the actual behaviour.

In view of the fact that notations for modelling behaviour have been out there for a quite a long time, it might appear a bit surprising that the use of behaviour models lags a bit behind in Model-based Software Engineering. There are different reasons for that. One of them is that the structural models that, typically, are used are class diagrams, which lack a natural mechanism for communication or for "hooking" in behaviour. The only mechanism they provide are method invocations – which are quite different from the communication mechanisms proposed by Hoare and Milner [21, 41].

In this report, we propose concepts and a modelling notation, which allows us to model the behaviour on top of structural models such as class diagrams. It is called the *Event Modelling Notation* (*ECNO*). The basic mechanism for integrating behaviour models with the structural models are *events*[1]. The *life cycle* of an object, basically, defines in which type of

[1] In Milner's terminology, our events would be called *actions*, and in Hoare's terminology events would be channels or channel names.

events the object can participate. We call this the *local behaviour* of the object which according to Harel and Marelly [18] would be the *intra-object behaviour*. ECNO could use any notation for defining the local behaviour of objects; but throughout this report, we will use a simple form of Petri nets, which we call *ECNO nets* for modelling the life cycles of objects. Due to their inherent notion of concurrent and parallel behaviour, Petri nets will have some slight advantage for modelling the local behaviour, which we will see much later in this report. The more exciting part is the coordination of different objects that need to join in the execution of the event. The ECNO provides *coordination diagrams* for defining which partners, in a given situation, need to participate in an event. We call the joint execution of events by different objects the global behavior, whereas Harel and Marelly [18] would call it inter-object behaviour. The mechanisms used in coordination diagrams are similar to the communication mechanisms of Hoare and Milner, but – as we will see later – coordination diagrams are more general in that many partners can be required, and more than one event might be jointly executed. We call the particular combination of objects and events that meet the requirements of the coordination diagram an *interaction*.

In order to get a better understanding of the basic idea of the Event Coordination Notation, we explain it by the help of a simple example.

1.1 Simple example: Workers and car sharing

As an example, we model a company in which *workers* are required to jointly do some *jobs*. Only if all the workers that are *needed* for the job join in, the job can be done. Our ECNO model of that company needs to make sure that the job can be done only when all the needed partners are ready for doing it. For simplicity, we assume that a job can be done instantaneously, once all partners join in. To make things slightly more interesting, we assume that the workers share *cars* for coming in for work and for leaving again. Therefore, the workers sharing the same care will come and leave together. And only when a worker is in, the worker can do a job.

This structure is shown in the structural domain model of Fig. 1.1 – as a kind of class diagram. The association between classes Worker and Car represent which workers share the same car. Note that at any given time, we assume that one worker can be partner in one shared car. The association between classes Worker and Job represents which workers are *needed* for a job. Every worker can be *assigned* many jobs – but a worker can only be active in one job at a time (we will come back to that later). Note that a single job may need more than one worker to participate. The class Job has an operation or method, which actually will be executed when the job is done – as in real live, executing a job will create more work: so the method creates other jobs (which is programmed in a classical way).

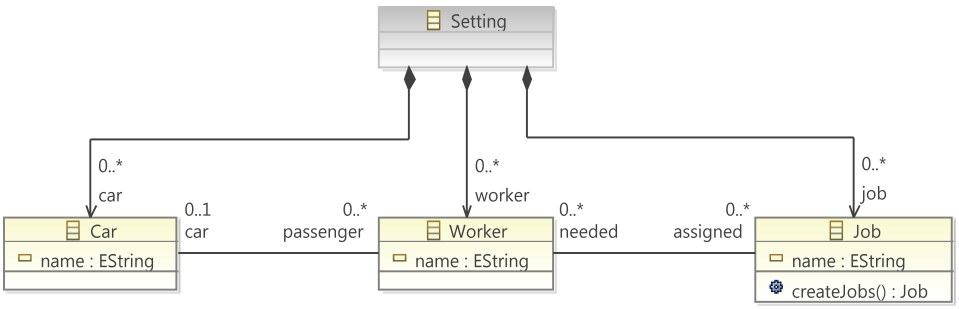

Figure 1.1: Structural domain model workers

The diagram from Fig. 1.1 has one additional class Setting, which contains a concrete configuration or setting of workers, jobs and cars. This, however, is needed for technical reasons only, so that the initial situation or configuration can be set up and contained somewhere and thus, be stored in some file. Fig. 1.2 shows an example of such a configuration as an object diagram, which is an instance of the class diagram of Fig. 1.1. In order to avoid clutter, we do not show the Setting object that contains all the objects.

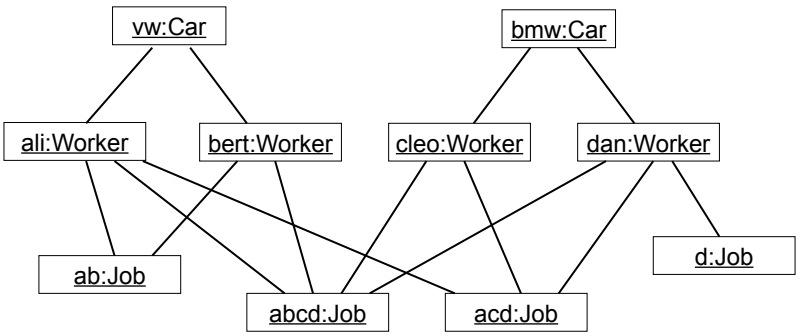

Figure 1.2: Some configuration

Up to now, we have defined the structure of our system by class diagram and its initial configuration by an object diagram. Some parts of the intended behaviour were explained in the text, but the behaviour was not modelled yet.

Next, we model the behaviour of the system. For modelling the behaviour we distinguish between the local behaviour of objects and the coordination of the local behaviour among the different objects. Before, we can actually coordinate something, there needs to be something that can be coordinated. In the Event Coordination Notation, it will be explicitly defined *events* which are coordinated. In our workers example, the events are arrive and depart, which mean that the workers and cars are arriving or

departing from the work site[2]; moreover, there is the event doJob, which means that a job is done; and there is an event cancelJob, which means that an existing job is cancelled.

These events are formally defined in the *coordination diagram* of Fig. 1.6, which defines the coordination of the behaviour among different objects. We start, however, with explaining the local behaviour – i. e. the life cycle – of the different elements first, and explain the coordination of the behaviour later. In ECNO, there are different ways of defining the local behaviour. Here, we use a simple form of Petri nets, which we call *ECNO nets*.

Figure 1.3 shows the ECNO net for the local behaviour of the Worker. The two places *home* and *work* represent the two states of a worker: being home or at work. Initially, the worker is at home. There are three transitions, which are annotated with assignments, where the right-hand side of the assignment refers to some event. These annotations are called *event bindings*. For now, it is only the reference to these events that is important. The transitions together with the event bindings define when a worker can participate in some event, and how the state of the worker changes when the worker participates in the respective event: When at home, the worker can participate in the event arrive, and when he does, he will be work after the event has happened. When the worker is at work, he can participate in doJob; and when he does, he still stays at work. When at work, he can also participate in the event depart, after which he will be home again.

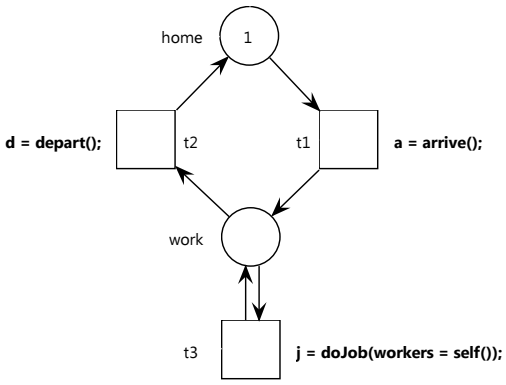

Figure 1.3: Behaviour of a Worker

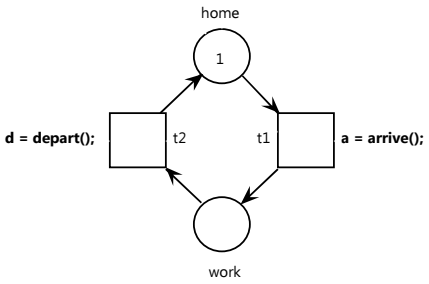

Figure 1.4: Behaviour of a Car

The ECNO net for the local behaviour of the Car is shown in Fig. 1.4. This is even simpler than the behaviour of a worker: A car can go round between *home* and *work* – alternately participating in arrive and depart.

The ECNO net for the local behaviour for the Job is shown in Fig. 1.5. After a job was created, it can either participate in a doJob event or a cancelJob event – after which no events are possible anymore. When the

[2]Note that we take the employer's perspective here for the meaning of arrive and depart.

job is done, it assigns itself (denoted by `self()`) to the *job* parameter to the event doJob – we will come back to that later. The transition t_1, which

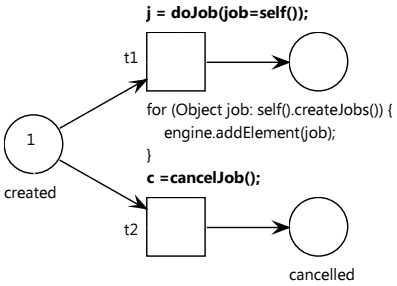

Figure 1.5: Behaviour of a Job

is bound to the doJob event, has another annotation right below it, which is called an *action*. This is a piece of Java code, which is executed if and when a Job participates in a doJob event. In our example, the action calls the method `createJobs()` of the job object (referred to by`self()`); as explained earlier, this method return a list of new jobs. The code snippets then iterates over all these newly created jobs to make them known to the ECNO engine (actually to its GUI).

The different examples of local behaviour show that the life cycle of some objects can be infinite, whereas the life cycle of others is finite.

Now that we know what the objects can do locally, we need to coordinate the behaviour of the different components. For example, we need to make sure that the workers sharing a car arrive and and depart together and with the same car; or all workers required for a job need to join the execution of the job. We could call this coordination *orchestration* if that term was not taken already [45].

Figure 1.6 shows the ECNO *coordination diagram* for our example. As mentioned earlier, this diagram defines the possible events, which are shown as rounded boxes. The coordination diagram also shows some parts of the structural model from Fig. 1.1 again, which are now equipped with some additional annotations. These annotations are explained below.

Let us assume that a Car can participate in an arrive event. This would of course require that the local behaviour of the car would be able to participate in the arrive event; in addition, the coordination diagram requires that all the workers sharing that car participate in this arrive event, too. To this end, there is a box with label arrive in the Car. This box, is linked to the reference passenger with an annotation arrive->ALL. The box is called a *coordination set* for event arrive of class Car. The annotation is called *coordination annotation* and says that, for a given car participating in an arrive event, every passenger (i. e. every worker at the other end of the links corresponding to passenger in the given situation) must participate in the arrive

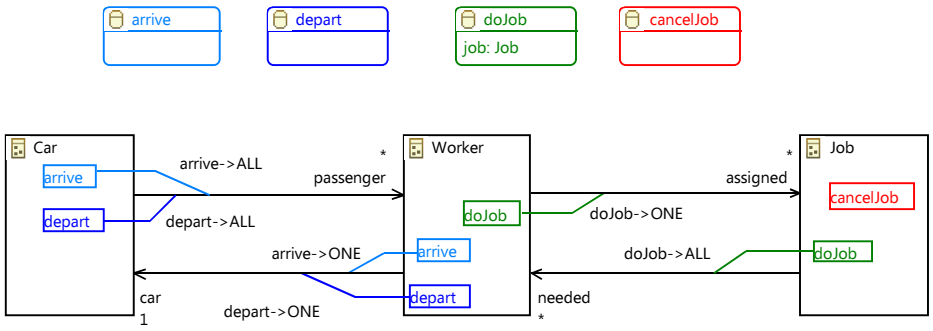

Figure 1.6: Coordination diagram

event too. For the car vw in the situation shown in Fig. 1.2, this means that workers ali and bert need to participate in the arrive event, too. Now, for a Worker participating in an arrive event, there is another coordination set for arrive which imposes additional requirements of other elements participating. The Worker has a coordination set which has a coordination annotation arrive->ONE linked to reference car. This means that for a Worker participating in the arrive event, there must be one Car at the end of the link car, which must participate in the arrive event. In our example, for ali as well as for bert, this car would be vw, which is – not by accident – the car we started our exploration from. So, we know that the workers ali and bert as well as the car vw could participate in an arrive event. We call such a combination of objects and events an *interaction*; and since all requirements by coordination sets of the involved objects and events are met, we call it a *valid* or an *enabled* interaction. A valid interaction can be executed, which means that all participating objects participate in executing the arrive event. If and when executed, the car vw as well as both workers ali and bert would be in their local states *work*, due to their local behaviour. After such an interaction, for these three objects, an interaction with the depart event would be enabled, by the coordination sets and coordination annotations for depart – with the same arguments as for the arrive event before. And there are similar interactions for cloe and dan who share car bmw for coming to work together.

Now let us assume that all workers are in for work (all their ECNO nets would be in state *work*) and the configuration is still as shown in Fig. 1.2. This means that, locally, each worker could participate in a doJob event. Let us assume cloe would want to participate in a doJob event: since there is a coordination set for event doJob for Worker, other object would be required to participate; the coordination annotation doJob->ONE would require one of the jobs assigned to cloe to participate in the doJob event too. In our example, there are two possibilities, job acd and abcd. Let us investigate job acd. Since the life cycle of this job is still in its initial state *created*, the

local behaviour would allow the job to participate in the doJob event. The coordination set of Job for event doJob with the coordination annotation doJob->ALL linked to reference needed, says that also all the workers at the end of the link need to participate in the doJob event; for the job acd, this would be the workers ali, cleo, and dan. The worker cleo is already participating in the doJob event – we started from there. But, now also ali and dan need to participate – and according to their local behaviour they can. For ali and dan we still need to check the coordination annotations, which require that one of the jobs assigned to them would participate in the doJob event. Actually, job acd is already participating; therefore, workers ali, cleo, and dan and the job acd participating in the doJob event would be a valid interaction. All requirements of the coordination diagram are met. After executing this interaction, the workers would still be in the state *work*, but the job acd would be in state *done*. And when the interaction is executed, the job acd would also execute the action attached to the doJob transition, which is creating some new jobs; this would actually create a new object in the configuration of Fig. 1.2, which shows that the configuration can change dynamically.

Up to now, we have seen the basic principles of ECNO and how events and coordination sets and annotations are used to define combinations of object and events that can be executed together as interactions. These requirements are local in the sense that each requirement for an object involves only objects to which the object is directly related by links in the current configuration. But for an interaction, the requirements for every object need to be met. In combination, all these local requirements can result in interactions that consist of many different partners and even many different events. Luckily enough, the valid interactions in a given configuration are computed by the ECNO execution engine fully automatically. Basically, the model or code generated from the models above can be executed without any additional programming. We will give a brief overview of the ECNO Tool in Sect. 1.3.

For now, let us have a look at one minor twist in our workers examples. As discussed before, we started with cloe participating in a doJob event, chose to let job acd participate too, then were forced to let ali and dan participate too. Then for ali and dan we still needed to find one Job to participate too. We chose to refer to the job that was involved already acd; but, we could have chosen or at least could have tried to choose another job, say abcd. Then, in the same interaction two jobs would be executed; but, we do not want to allow that. In our model, this is actually not allowed for the following reason: We had seen earlier, when discussing the local behaviour of Job on Fig. 1.5, that the job passes itself `self()` as a parameter to the doJob event. The local behaviour of each partner defines (in the event binding) whether to assign one or more parameters to an event or not. For example, the Worker does not assign a value to the parameter of the job in the ECNO

net of Fig. 1.3. In principle, different partners in an interaction can provide values to the same parameter of the same event; but in that case, ECNO requires that it is the same one (in the sense of Java's equal). If different partners assign different values to the same parameter of the same event, the interaction would be invalid. Since each job assigns itself as a parameter, this implies that only one job can participate in an interaction in our model. Of course, parameters can not only be used for guaranteeing uniqueness of some partners; parameters can also be used in the actions of the local behaviour – this way exchanging information among the participants of an interaction. This is discussed in more detail later in this technical report.

1.2 Basic concepts and terminology

In this section, we give an overview of the main concepts of the ECNO and the terms we use for them. For now, we focus on the terms which are at the core of ECNO and form the basis for explaining ECNO's more advanced concepts later in this technical report.

1.2.1 Object-oriented modelling

The ECNO is based on and extends object-oriented modelling. In particular, we use *class diagrams* and *object diagrams*. Here, we do not explain the concepts of class diagrams and object diagrams; we just name those concepts that ECNO uses and builds on. Since, most of our examples are based on the Eclipse Modeling Framework (EMF) [8], we stick to EMF's terminology.

Basically, we use *classes* and *references* between them along with their *multiplicity*. Later, we also use *inheritance* on classes.

For classes, we use *attributes*, but attributes are not specifically exploited by ECNO. In some of our class diagrams, we use *compositions*, which are a special case of references. Like attributes, compositions are not directly exploited in ECNO. Classes and references are defined in the scope of a *Package*.

We have seen an example of a class diagram or a package in Fig. 1.1 – actually it is an *Ecore diagram*, which is the EMF version of class diagrams. Though technically not quite correct, we call Ecore diagrams class diagrams throughout this technical report, except when discussing the technical details of a technology.

An *object diagram* is an instance of a class diagram, which shows a specific situation of a system. Figure 1.2 shows an example of an object diagram, which is an instance of the class diagram from Fig. 1.1. An instances of a class is an *object* and an instances of a reference is a *link* – or vice the type of a link of an object is a references of the class.

In the context of this technical report, we call class diagrams that are representing the concepts of an application's domain a *domain model*, and

the objective of ECNO and all our modelling is to bring our models as close as possible to *domain models* – and then generate executable code from them.

1.2.2 The Event Coordination Notation

The ECNO aims at modelling the behaviour of a domain on top of an object-oriented or structural part of the domain model. To this end, the ECNO extends the notion of objects and classes of object-orientation. In order to make the difference explicit, we call ECNO's objects *elements*, and we call ECNO's classes *element types*. In a sense, an ECNO element is an object with an explicitly defined *life cycle*. The life cycle of the element is defined by a model which defines the *local behaviour* for a specific element type. So, an element type consists of the class and the local behaviour, and some more concepts, which are discussed a bit later. In most of our examples, the local behaviour is defined by a special version of Petri nets, which we call *ECNO nets*. In our example, the the element types were defined by the class diagram of Fig. 1.1 and the ECNO nets of Figures 1.3–1.5, by using the same name for the class and the ECNO net.

The life cycle of an element defines, at which points in time the element could participate or engage in some *event*, and how the state of the element changes, when the element participates in the respective *event*. Actually, we need to distinguish between *event types*, and an instance or occurrence of an event at a specific time at runtime when an *interaction* occurs or is executed. The *event types* are defined in ECNO's coordination diagrams. The ECNO nets refer to these event types for defining the local behaviour. For simplicity, we call the instance of an event just *event* in the rest of this technical report – so an instance of an event type is an event. Note that even though the relation between an even and an event type is an "instance of" relation, which looks similar to the relation between objects and classes or elements and element types, the nature of events is fundamentally different from objects or elements. An event is inherently volatile and – at least conceptually – exists only for the time an interaction is executed: instantaneously. After that, all events evaporate; only their effects – defined by the life cycles of the elements participating in the events – stay.

An *interaction* is the joint participation of some elements in some events, which conceptually is executed instantaneously[3]. What constitutes a legal combination of elements and events is defined by the local behaviour of the element as well as the *coordination annotations*. As explained already, the local behaviour defines, whether an element could participate in an event at a given time. The coordination annotations define which combinations of elements and events are valid. Basically, each coordination annotation

[3]We will see later that interactions are executed transactionally; in particular, they are executed atomically and in isolation.

formulates a requirement of the following nature: if an element of some type participates in an even of some type, one or all elements to which there exists a link of a certain reference need to participate in that event too. We have seen in Fig. 1.6 that, for a Worker to arrive, the (one) car to which the worker is linked must participate in the arrive event too. Together, these requirements might require that many elements need to participate in a valid interaction – once one element participates in some event. And it might happen, that there is no such combination at all – in which case the overall behaviour would not allow the element to participate in that event at that time. We will see later, that the local behaviour of elements can require other events to join when executing an event – effectively synchronizing different events.

The coordination annotations are defined in an ECNO *coordination diagram*. On the side, coordination diagrams are used to define the event types, which are used in the coordination annotations as well as in the ECNO nets for the local behaviour.

As discussed above, in any given situation, the coordination annotations together with the local behaviour for the elements define which interactions would be possible. And the local behaviour and, in particular, the *actions* define what each element would do and how its state (including that of its local behaviour) would change when the interaction is actually executed. The question, however, is when are possible interactions executed? Actually the ECNO does not define that at all. The ECNO defines only which interactions can be executed (are valid) – and the ECNO execution engine will make sure that only valid interactions are executed. It is left to controllers on top of the ECNO engine to decide when valid executions are triggered or scheduled for execution. Typically, the execution of interactions is triggered by the user by clicking on some button in some Graphical User Interface (GUI), and the ECNO Tool comes with some predefined controllers and GUIs for that purpose. We briefly discuss this GUI in Sect. 1.3 about tooling. For realistic applications one would use the ECNO Framework for programming own controllers which are integrated into an own GUI, which is discussed later in this technical report.

For now, we can assume that, for some element types and some event types, which are specifically marked as GUI types, there are some buttons, which are enabled when some interaction for that element and event type are possible (see Fig. 1.10–1.12 for a glimpse of the GUI).

1.3 Tooling: Quick guide

Introducing a notation for coordinating interactions is one thing. The proof of the pudding, however, is bringing the notation into action and make it work. To this end, a tool along with a set of examples is published together

with this technical report: the ECNO Tool.

The ECNO Tool is independent from any specific technology and works together with any object-oriented technology. This independence of a specific technology is achieved by adapters, which will be discussed later in this technical report. Most of our examples, however, are based on the Eclipse Modeling Framework (EMF) and its Ecore models. In this section, we give a brief account of how to get started with EMF. This cannot replace a thorough introduction to EMF [8], though.

The ECNO Tool has its own graphical editor for ECNO coordination diagrams, which is briefly discussed in this section.

In order to support ECNO Nets, the ECNO Tool defines a new Petri net type for the ePNK [33]. In this section, we will briefly discuss how to create and edit ECNO nets, for more details on how to used the ePNK, however, we refer to the User's Guide of the ePNK Manual [35].

We explain the use of EMF and the ECNO Tool by going step by step through the Workers and Car Sharing example of Sect. 1.1. The best thing to do would be to do this example hands on. In that case, we suggest that you install the ECNO Tool and the example as discussed in the installation instructions in Appendix B.

1.3.1 Structural models

At first, we need to create the class diagram from Fig. 1.1. Technically, this is actually not a class diagram, but an *Ecore diagram*, which is a light-weight version of UML class diagrams following the principles of EMOF [44]. But, for the purpose of the ECNO, this need not bother us too much.

Once you have installed the ECNO Tool and imported the example project `dk.dtu.imm.se.ecno.example.workers` (see Appendix B), you should have a look into the folder "models" of this project, from which all the code can be generated fully automatically. The Ecore model for our example is contained in file "workers.ecore". This file, however, contains the model only; the actual diagram information is contained in "workers.ecorediag". If you double click on it, it will be opened in a graphical editor as shown in Fig. 1.7.

Later, you might want to create you own Ecore models. You can create them by right-clicking in some folder or project and by selecting "New → Other..."; in the opened "New" dialog, you should select "Ecore diagram" in the category "Ecore Tools". You can quickly find this by entering something like "Ecore" into the filter field at the top of the "New" dialog. Then you will be asked for a name and you need to follow through the steps of the wizard.

For now, you do not need to create a new Ecore diagram – you do not even need to change it. Note that, in this model, the operation `createJobs()` in class `Job` has a so-called annotation. This annotation carries the

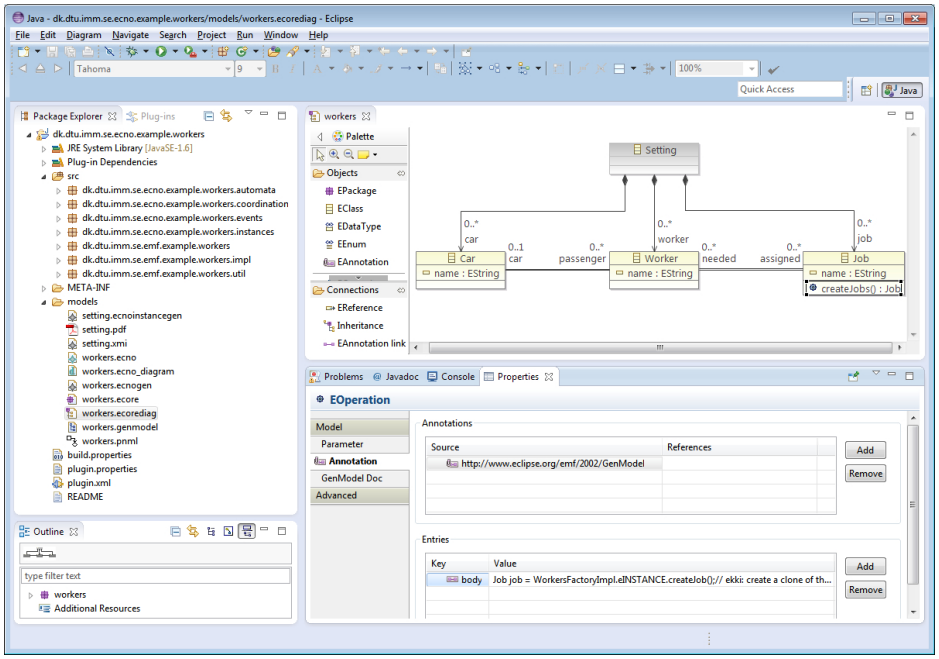

Figure 1.7: Workers example: Ecore diagram

body of the implementation of this operation. This way, it is possible, that the complete code can be automatically generated from the models.

In order to generate code from this model, we need to create a so-called *EMF generator model*. The generator file allows us to customize the code generation in many different ways. In our example, however, we have made one change only: the code generated from the Ecore model should go to the sub-package `dk.dtu.imm.se.emf.example`, which makes it easier to distinguish the code generated by EMF from the code generated by ECNO. In our example, the EMF generator file is "workers.genmodel".

In the workers example, the generator model exists already. If you want to create a generator model for a new Ecore model, you would select the file with the Ecore model (not the diagram), right-click and then select "New→Other..."; in the opened "New" dialog, you then need to select "EMF Generator Model" in the "Eclipse Modeling Framework" category, and follow through the dialog (you need to press the "Load" button, when asked for the model).

Once the generator model file is opened in the simple EMF tree editor, the code can be generated by a right-click on the top level element and selecting "Generate Model Element". This will generate all the code from the model that we need in our example[4]. The model code generated by EMF

[4]Readers who know EMF already might know that EMF can also be used to generate the so-called "Edit Code" and the "Editor Code"; for now, we do not need this, since we

can be inspected in the sub-packages in `dk.dtu.imm.se.emf.example`; basically, for each class of the model, there is an interface and an implementation class. Moreover, there are some utility and helper classes. Most importantly, there is a factory, which should be used for creating instances of the classes (see [8] for more details).

1.3.2 Coordination diagram editor

Next, we discuss the ECNO coordination diagram from Fig. 1.6. Like for Ecore models, the actual model and the diagram information are split up into two files. The ECNO coordination model is contained in the file "workers.ecno", whereas the diagram information is contained in "workers.ecno_diagram". The diagram can be opened by double-clicking on "workers.ecno_diagram", which will look as shown in Fig. 1.8.

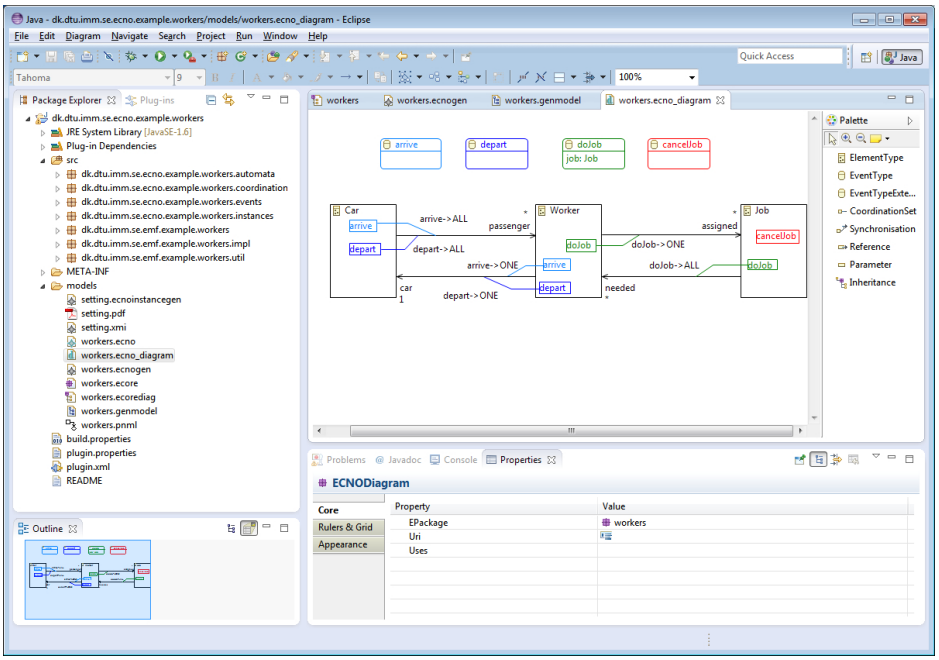

Figure 1.8: Workers example: ECNO Coordination Diagram

As you can see in the properties view at the bottom, the ECNO coordination diagram has a reference to the Ecore models "workers" from Sect. 1.3.1.

For now, we do not need to create a new coordination diagram. If you want to create a new ECNO coordination diagram, this can be done in a way similar to the Ecore diagram: Once the project or folder to which the diagram should be created is selected, right-click and select "New→Other...";

use our code in a simple Java application only in this overview.

in the opened "New" dialog, you should select "ECNO Coordination Diagram" in the category "Examples"[5]. You can quickly find this by entering something like "ECNO" into the filter field at the top of the "New" dialog. Then you will be asked for a name and you need to follow through the steps of the wizard. Once the editor with the newly created ECNO Coordination Diagram is opened, the first thing to do is selecting the underlying Ecore model. To this end, you need to make the Ecore model available in this editor, which is done by right-clicking into the empty canvas of the editor, and selecting "Load Resource"; in the opened "Load Resource" dialog, you can select "Browse Workspace..." and select the file with the Ecore model ("workers.ecore" in our example). Once you have loaded the resource, you can go to the properties view, and select the `workers` package for the EPackage property.

When a package is selected in this way, the ECNO coordination diagram editor will automatically create element types for each class that it finds in the Ecore package and connect the element type to the resp. class. The ECNO coordination diagram editor will also create the references between the classes of the selected package and the inheritance relations between them. This make it easier and faster to create an ECNO coordination diagram from an existing Ecore diagram. In many cases, you do not need element types for all classes; you can delete them manually. For our workers example, we have deleted the class `Setting` since this does not have any behaviour; it just servers as a container for all the other elements. From this basic skeleton, you can then create coordination sets, and connect them to the references by synchronisations with the tools that are available in the editors tool bar. And you can create event types and their parameters.

Note that, in the ECNO Coordination Diagram for the workers example, we have used different colours for different events. This was done manually, however, for the sole purpose of making it a bit easier to read the ECNO Coordination Diagrams. Colours do not have any meaning.

From the ECNO coordination diagram alone, we cannot create any code. We first need to create the models for the local behaviour and – similarly to EMF – create an ECNO generator model.

1.3.3 ECNO net editor

Next, we discuss how to create the ECNO nets of the workers example shown in Figures 1.3–1.5 of Sect. 1.1. The ECNO nets of the workers example are contained in the file "workers.pnml", which is shown open in Fig. 1.9.

Since ECNO nets are implemented as a Petri net type of the ePNK [33], creating and editing of ECNO nets follows the rules of the ePNK, which can be found in the User's manual part of the ePNK manual [35].

[5]This category will eventually be changed to ECNO.

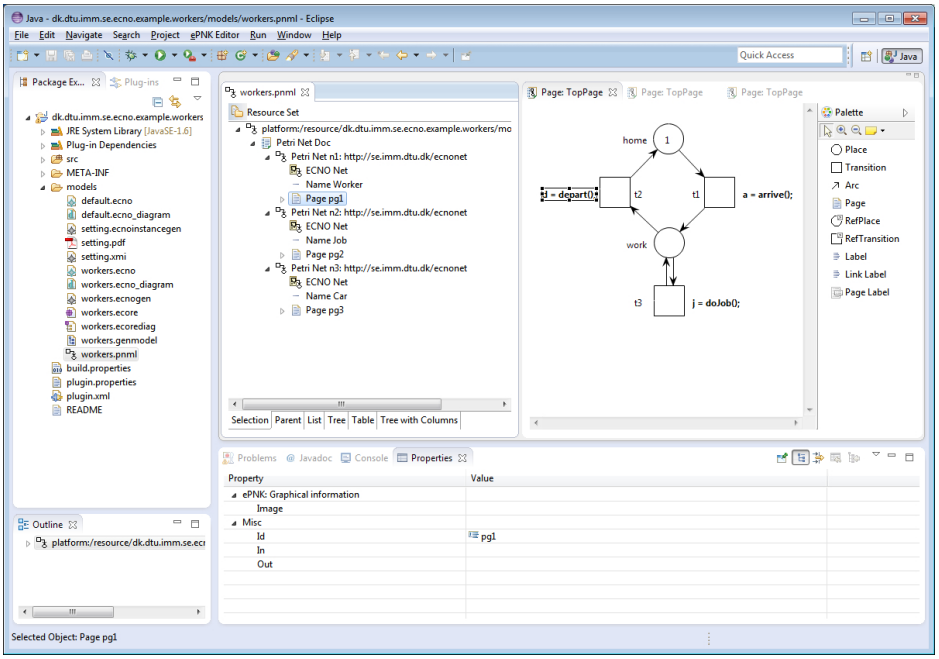

Figure 1.9: Workers example: ECNO Nets

Here, we give a brief overview of how to use the ePNK editor for inspecting existing ECNO Nets, and how to create and edit new PNML documents with ECNO nets. As for all resources, the ePNK editor can be opened by a double-click on the resp. file. Note that the ePNK will always open a tree editor for a PNML document, which is the primary editor. In Fig. 1.9, the ePNK tree editor is open on the left side. When a PNML document is initially opened, this ePNK tree editor is the only one open and only the top-level element is visible. From there, the tree elements can be browsed and inspected in the usual way. By double-clicking on a page, a graphical editor for the ECNO net is opened[6]. Normally, this graphical editor is opened on top of the tree editor; in Fig. 1.9, the graphical editor has been moved to the side, so that both, the tree editor as well as the graphical editor, are visible. The places, transitions and arcs can be created by using the tools from the toolbar of the graphical editor. The creation of *labels* for the event bindings, conditions, and actions is specific for the ePNK (for details see Sect. 3.4.2 of the ePNK Manual [35]): you first need to create a label with the respective tool from the toolbar; then, the label needs to be linked to an element by the "Link Label" tool, which is dragged from the label to the respective element; only then, the ePNK will offer you to choose the type of the label. Global labels for import statements or attribute declarations

[6]Note that the tree editor should not be used for making any changes withing a page. To this end, the graphical editors should be used.

are added as so-called *page labels*. When created by the respective tool from the tool bar, the ePNK will ask which kind of label (import or attribute) it should be. The legal syntax for the different labels will be explained, in the more detailed discussions later in this technical report. For now, just follow the examples.

Note that the graphical editors of the ePNK for the individual pages of the net run as clients of the tree editor. Therefore, all save operations on a PNML document need to be done from the corresponding tree editor. The graphical editors do not even show the dirty flag, when the PNML document file is changed.

A new empty PNML Document can be created by selecting the folder or project to which the file should be added, then selecting "New→Other..." and, in the "New" dialog selecting "PNML Document" in the category "ePNK". In this empty document, you can create new nets, by right-clicking the "Petri Net Doc" and selecting "New Child→Petri Net `http://se.imm.dtu.dk/ecnonet`", which represents ECNO nets. In the same way, you should create the child element for the name of the net and its top-level Page. In principle, the ePNK allows a net to have more than one page; since ECNO nets, typically, are quite small, we would recommend to use ECNO nets with a single page only.

Once you validate ECNO nets (right-click and select "Validate" on an item) you might realize that every Petri net element is supposed to have a unique identifier (which is required by the PNML standard [23, 20]). You can automatically add the missing identifiers by double-clicking on the top-level "Petri Net Doc" element.

1.3.4 ECNO code generation

Once we have created the ECNO coordination diagram and the ECNO nets, we are – almost – ready to generate code. Similar to the EMF technology, ECNO has a ECNO generator model for adding some configuration information. In our example, this *ECNO generator model* is contained in file "workers.ecnogen".

A new generator model can be created by using the "Ecnogen Model" wizard, which can be – as usual – started with "New→Other...". The ECNO generator model, consists of a single element for which the following properties can be set:

Ecno Model The reference to the ECNO coordination model.

Behaviour Model The reference to the PNML Document that contains all the ECNO nets with the local behaviour of each element type (note that element types for which no behaviour is defined will have a simple default behaviour).

Emf Gen Model The reference to the EMF Generator model, from which the model from the Ecore model was generated; it is possible to provide a reference to several models here, in case different ECNO packages and Ecore packages are used. The first reference, however, should be the one to the EMF generator model for Ecore model underlying the ECNO coordination model above.

Model Class Name The name of the generated class that represents the ECNO coordination model.

Automata Factory Class Name The name of the factory class, which will be created by the code generator. This factory class will be used by the ECNO engine for creating the behaviour for new elements.

PackageAdapter Factory Class Name (optional) If the ECNO model should be registered as an extension with Eclipse, this is the name of the respective factory class. This is not relevant, if the generated code is run as Java code only; it is relevant when the ECNO model should be run inside an Eclipse application, which is discussed later in this technical report.

Base Path ECNO Automata The base package (path) to which all the code for the automata for the local behaviour is generated.

Base Path ECNO Events The base package (path) to which the code for the events (actually the event values class) is generated. This class provides the access to the values of the involved events in event bindings, in conditions, and in actions.

Base Path Model Class The base package (path) to which the class representing the coordination diagram is generated.

Required This is referring to a list of ECNO gen models for the ECNO models on which this ECNO model depends on. For models that consist of a single ECNO model only, this attribute is empty.

Some of the above properties are strings only and can be entered as strings. The references to other resources are not strings, however. In order to create these references, the resources containing these elements need to be loaded by the "Load Resource" mechanism, which was discussed in Sect. 1.3.2. Once the resources are loaded, they can be selected by a drop down menu in the properties view of the ECNO Gen Model editor.

In order to generated the code, the ECNO generator model should be opened in the ECNO generator model editor. Then right-click on the top-level element and select "ECNO→Generate ECNO Package code". In our workers example, this generates the code in the sub-packages `automata`, `coordination`, and `events` in package `dk.dtu.imm.se.ecno.example.workers`, which is the code which will be running in the ECNO engine.

1.3.5 ECNO instance code

In order to start executing something, we need to have some start configuration, such as the one shown in Fig. 1.2 for our workers example. For creating the initial situations, we make use of an generic editor for instances of Ecore models, which is called *dynamic instance editor* and is part of EMF.

In order to create such a configuration, we can open the Ecore model in the EMF tree editor, select an element, right-click and select "Create Dynamic Instance...". In our workers example, we have created a dynamic instance from the class `Setting` of "workers.ecore"; this instance is contained in file "setting.xmi", which exactly represents the configuration from Fig. 1.2. Once a dynamic instance is created, the instance can be edited as in any other EMF tree editor. Therefore, we do not discuss the details of this editor here.

From this instance, ECNO is able to generate a class, which creates this instance and starts the ECNO engine on it. For adding some configuration information, there is yet another configuration file, which we call *ECNO instance generator model*. This file can be created with the "Ecnoinstancegen Model" wizard.

Similar to the ECNO generator model, the ECNO instance generator model contains a single element with references to the instance, to the ECNO generator model, and with the name and the package for the instance class to be generated.

When the top-level element is selected in the ECNO instance generator model, right-clicking and selecting "ECNO→Generate ECNO Instance Code" will generator the Java class with the instance code. In our workers example, the class `Setting` is generated in the package `dk.dtu.imm.se.ecno.example.workers.instances`. This class has a static `main()` method, which allows us to start it as a Java application.

1.3.6 Running the example

At last, we can start the generated code from the generated instance class `Setting` above. This is done as usual for Java applications in Eclipse by right-clicking on the class `Setting` and then selecting "Run as→Java Application".

This will open a window as shown in Fig. 1.10, which shows all the elements of the initial setting[7]; the enabled buttons show for which event types there is an interaction for that element enabled. When the mouse is moved over an enabled button, the tool tip information shows the elements, events and event parameters that are involved in this interaction – once it is selected for execution. When a user clicks on an enabled button, the respec-

[7] Actually, it shows only the element types, which are marked as GUI elements in the model.

tive interaction is executed, and all buttons will automatically be updated depending on the interactions that are enabled in the new situation. If, for example, the user presses the "arrive" button for the element [1] (the car VW) in this situation, the situation changes to the one shown in Fig. 1.11. If the user then presses the "doJob" button on element [7], the situation of Fig. 1.12 will be reached.

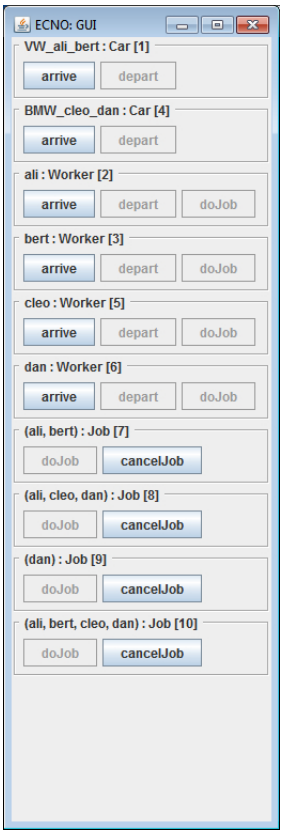

Figure 1.10: Initial situation

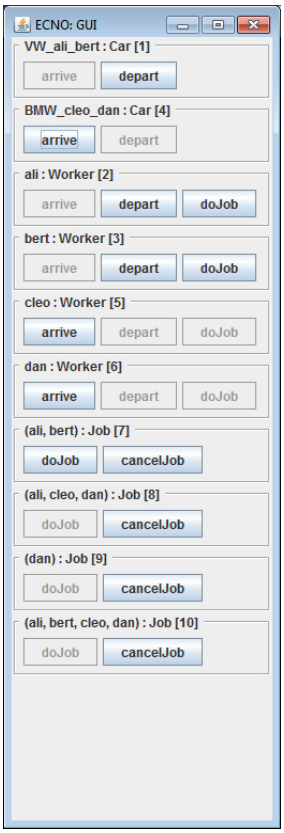

Figure 1.11: After pressing "arrive" on [1]

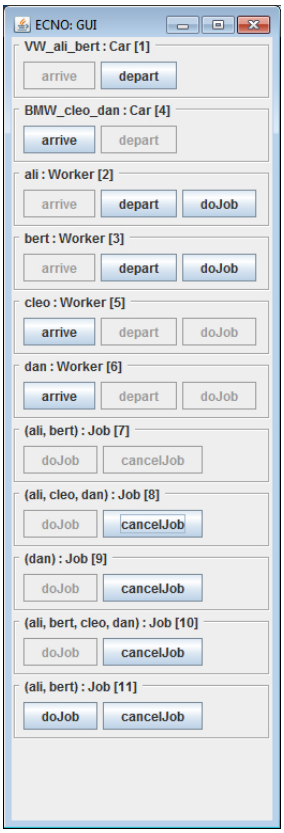

Figure 1.12: After pressing "doJob" on [7]

Since the *ECNO GUI* is generic and works for all ECNO models in the same way, it does not look too nice. But, it serves its purposes for experimenting with ECNO. For realistic applications, ECNO provides a framework for implementing customized GUIs, which will be discussed in Sect. 5.5.2 – continuing the workers example of this informal introduction.

The GUI will also indicate if, for some element and event type, there is more than one interaction enabled that involves the resp. element and event type. This is indicated by an asterisk for the respective button; the user can iterate over all enabled interactions by pressing the mouse button together with the "SHIFT" key. Once the last possible interaction is reached, this is

indicated by an exclamation mark. Pressing the mouse button together with the "CONTROL" key, will reset the iteration of the possible interactions to the first one again (the order might vary though in different iterations).

If the user presses the exit icon of the GUI window, the ECNO Engine terminates. Actually, the ECNO does this in a slightly more sophisticated way: The GUI works as a so-called engine controller; the ECNO engine terminates automatically, when the last controller terminates. The details are discussed later, when discussing how to use and program own controllers (see Sect. 5.5 for details).

1.3.7 Overview of advanced features

The sections above gave a brief overview of the most important features of the ECNO Tool. We did not discuss some more advance features, which are important for running more realistic applications. In Chapter 5, we discuss all the basic features as well as the advanced features of the ECNO Tool in more detail. Below, we just mention some of the more advanced features.

In ECNO, it is possible to save the current state of an ECNO application and to later start it in that state again. This way, ECNO supports some basic persistence mechanism. Moreover, the code generated from ECNO models can be run as an Eclipse application (in the example above, we run it as a Java application only). When an ECNO application is run as Eclipse application, there is a view available, which gives an overview of all running ECNO engines; this view, allows (dependent of the underlying object-oriented technology) not only to save the current state of an application, it also allows to undo interactions in reverse order and to redo them again. Moreover, this view gives access to some statistics on the computation of possible interactions and their executions – mostly for experimentation purposes.

The standard GUI of an ECNO application is useful only for testing an ECNO application and for quite small examples. For realistic applications, the developer would need to develop a customized GUI. The ECNO Tool provides a framework for implementing such customized GUIs.

Maybe, the most important feature for the practical use of ECNO is its mechanism to build ECNO applications from different parts, so that the ECNO models do not need to be monolithic. To this end, ECNO has a notion of packages. What is more, it is possible to integrate different ECNO packages that are based on different underlying object-oriented technologies. This way, ECNO might eventually ease the integration of systems with different technologies. In Sect. 4.1, we discuss a simple example, which is built up from different packages. The more technical details are discussed in Chapter 5.

1.4 Literature

The ideas of ECNO have evolved over many years and started out from a meta model that distilled the essence of business process modelling notations: AMFIBIA [2]. For capturing the behaviour of these concepts, AMFIBIA used an ad-hoc notation for the local behaviour and for the coordination of the behaviour of the different elements. The ad-hoc notation that was used in AMFIBIA was later formalized and implemented in a kind of pre-cursor of ECNO, which we called MoDowA [49, 38]. The core concepts of ECNO go back to the ad-hoc notation of AMFBIA and MoDowA – ECNO is more general and the too tight integration with aspect-orientation was dropped, so that ECNO – at its core – is not explicitly aspect-oriented anymore. Moreover, inheritance was introduced for elements and for events, which needed some careful tuning; for that reason, there is a complete chapter devoted to the discussion of inheritance in ECNO: Chapter 4. Starting out from the challenges of behaviour modelling [30], we then defined the core concepts of ECNO [32, 37, 36] with minor variations, which now seem to converge.

As pointed out in our earlier work [37] already, none of the concepts used in ECNO are particularly new or original; the contribution of ECNO is more in the combination of its concepts and, on the technical side, its integration with existing object-oriented technologies.

ECNO's coordination mechanism via events resembles the synchronization of actions in process algebras [21, 22, 41]. One difference, though, is that ECNO's synchronization is not restricted to bi-lateral synchronizations and that the required synchronizations might dependent on the dynamically changing underlying structure of the system. But also this aspect has been seen before in process algebras like ACP [4], the chemical abstract machine [5], or the Π-calculus [42]. What is new, however, is that, in ECNO's coordination mechanism, different of these synchronization mechanisms work together, combining these coordination requirements transitively, which allows us to define much more complex interactions.

Another major concern in the design of ECNO was the clear separation between coordination aspects and computation aspects of a system. Actually, ECNO is about coordination only, but ECNO's concept of actions provides a way to interface with the computational aspects by invoking methods or functions. This idea, however, is not new either: Harel and Pnueli [16] had proposed the distinction between transformational and reactive systems. ECNO takes care of the reactive aspect of the system by defining possible interactions – the transformational aspect is left to the underlying programming language (Java in our case) for the actions by invoking methods.

Another major concern of ECNO is the distinction between local behaviour and global behaviour [30]. But also this idea is not really new: Harel

and Marelly [18] distinguish between intra-object behaviour and inter-object behaviour, which correspond to local and global behaviour, respectively. The only difference is the way this behaviour is represented. Concerning the local behaviour, this is mostly a question of syntactic sugar. For inter-object behaviour (global behaviour), Harel and Marelly use a set of Live Sequence Charts (LSCs) [11], which are an extension of Message Sequence Charts [24]. This is a scenario-based and temporal approach, where the focus of inter-object behaviour is the behaviour over time. In ECNO, the coordination annotations refer to the needed partners for a single interaction only: it is about behaviour at a time. Therefore, both approaches have a different focus. It might be interesting to combine both of them; this might in particular be interesting since ECNO does not have a way to define what must happen in a system – it defines what can happen only. LSCs [11] allow to characterize both kinds. But a detailed investigation of such a combination would require further research.

As discussed above, the ideas of ECNO started out from an ad-hoc notation in which aspects were an explicit modelling concept and therefore, ECNO has some relation to aspect-oriented programming [28, 40] or aspect-oriented modelling [7, 9]. Actually, from the philosophical angle the original ideas were close to the Theme approach [10] and closer to the idea of subject-oriented programming [19]. Anyway, the explicit notion of aspects was removed in ECNO again. A bit of the original subject-oriented idea survived in one of the two different concepts of inheritance on event types, which is discussed in Sect. 4.2.3. And by using some specific modelling patterns, ECNO can be used for modelling in an aspect-oriented way.

In a way, events of ECNO can be considered to be join points of AspectJ [27]. The difference, though, is that events are an explicit modelling concept [12], whereas join points are formulated on top of a program. This way, events are a concept of the domain, whereas join points are programming artifacts (which of course could have a counter-part in the domain). The coordination annotations of ECNO then correspond to pointcuts. Though stripped of an explicit notion of aspects, ECNO still shares some philosophy with aspect- or subject-orientation: joining events together via coordination annotations into interactions.

The local behaviour of elements could be modelled in many different ways. We could use traditional automata or StateCharts [17]. We mainly use a special form of Petri nets [46, 47], which we call ECNO nets. The reason for using ECNO nets was mostly a practical decision, since we could use our own framework for Petri net tools, the ePNK [31, 35], for easily implementing a graphical editor for ECNO nets. And the ePNK is based on EMF [8], which is the object-oriented technology that happens to be the default object-oriented technology of ECNO. It turned out to be useful that Petri nets have a natural notion of concurrent or parallel firing of transitions, when it comes to parallel behaviour (see Sect. 2.2.4 for details). Therefore, simple

automata are not sufficient for modelling the local behaviour of elements. Like Petri nets, StateCharts have a notion of parallel local behaviour, which makes them a good candidate for local behaviour, too. Our main concern with StateCharts would be that they might be too powerful: modellers might be tempted to put too much into the local behaviour of elements, since StateCharts allow nested complex states. But, this is up to future evaluation and a question of methodology, which is yet to be worked out in full detail.

At last, ECNO has some similarities with *agent-based software engineering* and *Multi-Agent Systems* (MAS) [57, 25]; but, at least in its basic form, ECNO would probably not qualify as an approach towards agent-based software engineering. This, however, depends on which level we look at things: From our point of view, ECNO is more a notation and technique[8] whereas agent-based software engineering is more a way of thinking. Anyway, some of the principles underlying ECNO were proposed by the proponents of agent-based software engineering. The two most important are: getting rid of the thread-oriented way of thinking, and giving agents control over what they do or to which kind of request they react – or as we would say in ECNO – in which events they participate. In addition, in agent-based software engineering, agents have attitudes and are pro-active and take initiative. Even disregarding the more social notions of initiative and attitude, ECNO elements are not even active – remember that ECNO models describe what can happen in a given situation, but they do not describe what must happen. Therefore, ECNO's elements are technically not agents. But, by adding controllers on top of elements (see Sect. 5 for details), elements can be turned active. This way, ECNO might be a notation and technique in which agent-based designs or agent-based thinking can be formulated and implemented. But, this is up to others to judge.

Speaking of agents, we should mention another approach, which uses Petri nets for defining local behaviour: Renew [39]. Renew also uses a mechanism for synchronizing different parts of a system with each other following some fixed relations between these parts. But, theses synchronisations need to follow some very specific containment structures following the so-called nets-within-nets paradigm [53]. By contrast, ECNO models can exploit the dynamic structure of the underlying object-oriented model for defining the required partners, which was one of its express goals.

Altogether, ECNO has many different flavours. On a first glance, ECNO might appear as just another process algebra, just another notation for aspect-oriented modelling, just another agent-based approach, just another form of transactions, just another ... And there is some truth to it. But, we believe that it is the combination of these different things and a carefully

[8]The methodology part of this technique is yet to be worked out in detail.

adjusted set of concepts that makes ECNO what it is: A way of clearly separating coordination from computation, and of separating coordination from local behaviour.

1.5 Overview of report

In this chapter, we gave a brief and informal introduction to ECNO. In the rest of this report, we discuss the details of the concepts and notations of ECNO and its use.

In Chapter 2, we discuss the core concepts of ECNO and motivate the choice of these concepts. In Chapter 3, we formalize the semantics of the core concepts of ECNO.

In Chapter 4, we discuss inheritance in ECNO. This concerns the behaviour of the elements, since all types in an element's type hierarchy contribute to the element's life cycle. More importantly, there is also inheritance on event types; actually, there are two different notions of inheritance for event types. Therefore, we devote a complete chapter to the discussion of inheritance in this report.

In Chapter 5, we discuss the use of the ECNO Tool for modelling, for generating the code, and for setting up and using initial configurations of ECNO applications. In addition, we discuss how to use the ECNO framework for programming own controllers, GUIs, and for programmatically interacting with an ECNO application.

In Chapter 6, we discuss two more examples. In particular, we discuss one larger example, a model of a workflow engine.

At last, in Chapter 7, we discuss what is achieved with ECNO so far, but also some of its current technical and conceptual limitations. From there, we discuss the next steps in the research on and development of ECNO.

Note that, this report also contains an appendix with a glossary of the most important terms of ECNO (Appendix A) and an appendix with installation instructions for the ECNO Tool (Appendix B).

Chapter 2

Coordination and Interaction

Chapter 1 gives a rough and informal account on the main concepts of ECNO. In the following, we give a more detailed account of ECNO's main concepts along with a precise meaning of these concepts. In this chapter, we focus on ECNO's concepts for coordination and interaction; to this end, we completely ignore ECNO's concepts for inheritance, which is a topic in its own right and is discussed in Chapter 4.

In order to explain some of the more advanced features, we discuss an other example, which formalizes the semantics of Petri nets [34]. The main reason for using this example, is that it makes use of most of ECNO's core concepts, but does not make use of inheritance. But, there are some additional reasons for choosing this example: First, this Petri net example continues the *Model-based Software Engineering* story that we started telling some year's ago [29], where the focus was on generating code from structural models and for standard functionality such as graphical editors with the *Graphical Modeling Framework* (GMF) [14]. Moreover, we discussed as to why modelling non-standard functionality and non-standard behaviour was still a challenge. The example that we had used at that time was called "A Petri Net Editor in 15 Minutes" – with the focus on the structural models and a graphical editor [29]. Now, we go beyond the graphical editor and show how also the behaviour of Petri nets can be modelled by using ECNO. Second, formalizing the semantics of a formalism such as Petri nets might give a hint already that ECNO could eventually be used to formalize its own semantics. Third, we need to understand the semantics of Petri nets anyway, since Petri nets form the basis of ECNO nets, which are our prevalent notation for modelling the local behaviour of elements.

We start with informally introducing Petri nets and then formalizing their syntax and semantics by EMF and ECNO. Later, we analyse and discuss the used modelling constructs of ECNO in more detail.

2.1 Petri nets

Petri nets are a formalism for modelling the synchronization and coordination of dynamic systems. There are many different versions and variants of Petri nets. Here, we use one of the most basic versions of Petri nets, which focus on the basic machanisms of synchronisation and are called *Place/-Transition systems* or *P/T-systems* for short [48].

2.1.1 Example and concepts

Figure 2.1 shows a simple example of a Petri net, which models the mutual exclusion of two processes by a semaphor. The Petri net models two agents

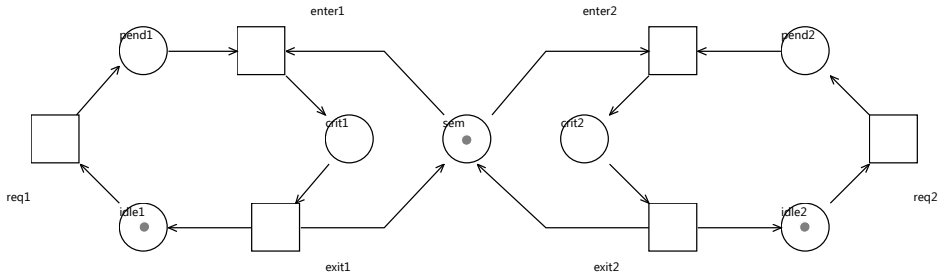

Figure 2.1: A Petri net modelling mutual exclusion

or processes, which cyclically run through the phases *idle*, *pending* (*pend*), and *critical* (*crit*). As indicated by the name, the two processes should never be in their critical section at the same time. This is achieved by each agent acquiring the *semaphor* (*sem*) when entering the critical section. The semaphor is returned again when the agent exits the critical section. In a Petri net, the possible states are represented by *places* which are graphically shown as circles or ellipses. A black dot, called a *token*, on a place indicates that the agent currently is in this state. In P/T-systems, it is possible that there is more than one token on a place, but this situation does not occur in our example. Figure 2.1 shows that, initially, both processes are idle (represented by the tokens on places *idle1* and *idle2*) and that the semaphor is available (represented by the token on place *sem*). A distribution of tokens on the places of a Petri net is called a *marking* of the Petri net, which represents the current state of the system.

In a Petri net, the possible state changes are modelled by *transitions*, which are graphically represented by squares or rectangles. The *arcs* from a place to a transition indicate on which places there needs to be a token for the transition to be *enabled*. In the initial situation of our example from Fig. 2.1, for example, the transition *req1* is enabled, since there is a token on the place *idle1*, which is the only place with an arc to transition *req1*. Likewise, transition *req2* is enabled since there is a token on *idle2*. All

the other transitions are not enabled in the initial situation. An enabled transition can *fire*; if and when a transition fires, it removes one toke from each place from which an arc is pointing to the transition; at the same time the transition adds one toke to each place to which it has an arc pointing to. The set of places with an arc towards the transition is called the *preset* of that transition. The set of places with an arc from that transition is called the *postset* of that transition. With the notions of preset and postset, the enabledness and firing of a transition can be formulated as follows: A transition is *enabled*, if every place in its preset has at least one token. When an enabled transition *fires*, one token is removed from each place in its preset and one token is added to every place in its postset.

For example, transition *req1* will remove a token from place *idle1* and add one token to place *pend1* when it fires. The transition *enter1* removes one token from place *pend1* and one token from place *sem* and adds one token to place *crit1*. Since both *enter* transitions need a token on place *sem*, it is guaranteed that never both processes are in their critical section at the same time, since the firing of a transition of a Petri net is atomic and instantaneous.

2.1.2 Playing the token game with the ECNO Tool

If you want, you can play a bit with the Petri net from Fig. 2.1 to get some feeling for the semantics of Petri nets – which nicely fits with its name: *token game*. If you don't want to play the token game, you can skip this subsection and safely read on in Sect. 2.1.3. You can obtain this example Petri net by importing the project `APetriNetEditorIn15Minutes.runtime` (as source project) to the workspace of your version of Eclipse as discussed in Appendix B.3. You will find the Petri net as file "semaphor.behaviourstates" in the folder "run" of this project, this file is a so-called *start configuration* for the ECNO engine, which defines in which way an ECNO application should be started and on which data it should be run. This configuration file also contains the states (in this example the initial states) of the local behaviours of all the elements – which is the reason for using the file extension ".behaviourstates". In this example, the actual elements are the objects of the Petri net that is contained in the file "semaphor.pertinets"; the file "semaphor.petrinets_diagam" contains the actual diagram information of the Petri net for the graphical editor.

You can start the execution of the ECNO engine by right-clicking on the file "semaphor.behaviourstates" and selecting "ECNO→Start ECNO Engine". Then, you will see a GUI popping up, with a button corresponding to each transition as shown in Fig. 2.2; in addition, the graphical editor opens – showing how the marking of the Petri nets changes when transitions are fired. Figure 2.2 shows the Eclipse workbench with one simulation engine running after firing some transitions. The enabled transitions of the

Petri net show up as enabled buttons on the GUI panel on the left; and clicking on the button of an enabled transition will fire the transition. When you click the button corresponding to an enabled transition, the enabled transitions are updated in the GUI panel, and the new marking is shown in the graphical editor.

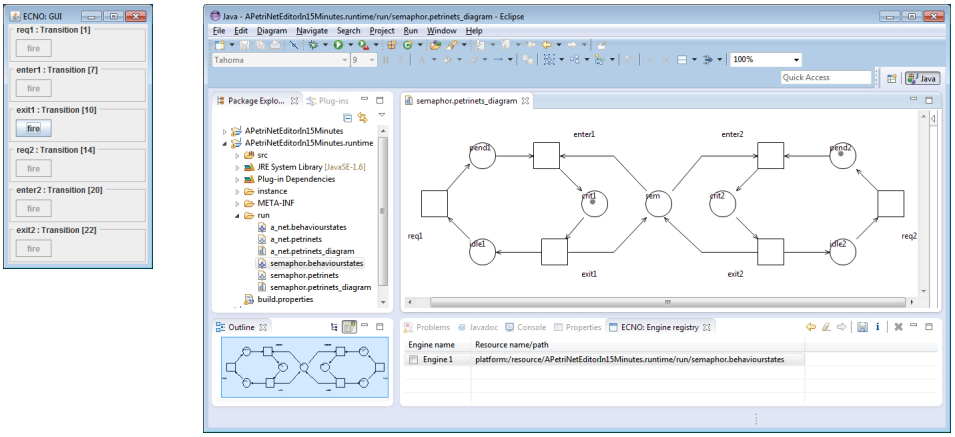

Figure 2.2: Playing the token game with the ECNO engine

In order to control this simulation and eventually terminate it, you should open the ECNO: Engine Registry view of ECNO Tools. You can do this by selecting "Window→Show View→Other..." and then selecting "ECNO: Engine registry" from category "ECNO". In this view, you will see all the currently running ECNO execution engines. By checking them (in the checkbox in the first column) and then clicking "delete" (the red cross on the top right of this view), you can shut down (delete) these engines.

If you have selected a row in the "ECNO: Engine registry view" by clicking on it, you can also get some extra information on the running execution, and you can redo and undo the previous or next interaction. You can also save the current state of the execution engine (it will be saved in the file from which it was started, which is shown in the third column of this view). If you start the engine next time from that file as discussed above, it will start exactly from the situation where you saved it. You can save the state by clicking on the save icon at the top right of the "ECNO: Engine registry view" – if a row for an execution is selected.

Note that, if you have started the ECNO Engine as an ECNO application, the ECNO Engine does not close automatically when you close its GUI or the editor showing the Petri net. You need to close the ECNO engine explicitly by deleting it via the ECNO registry view as discussed above.

Note also that it depends on the underlying object-oriented technology whether the state of ECNO application can be saved. But, as long as you stick with EMF and ECNO nets – as we do in this report for most of our examples – you can save the state of the ECNO execution engine. For a

selected ECNO execution engine, the save button is enabled only, if the underlying technology allows saving the state.

2.1.3 Formalizing Petri nets

Next, we formalize the syntax and semantics of Petri nets. We do not formalize Petri nets in the traditional way by using mathematical definitions [48]; we formalize Petri nets in a software engineering way by providing models [29, 34]: We formalize the syntax of Petri nets by providing an (Ecore) model for Petri nets with some additional OCL constraints; we formalize the semantics of Petri nets using ECNO. This formalisation was actually running behind the scenes when playing the token game as discussed in Sect. 2.1.2.

2.1.3.1 Abstract syntax of Petri nets

Note that we do not formalize the concrete (or graphical) syntax of Petri nets here, which would correspond to modelling a graphical editor for Petri nets (see [29] for more details). We define the abstract syntax of Petri nets only, basically formalizing the concepts of Petri nets and how they relate to each other.

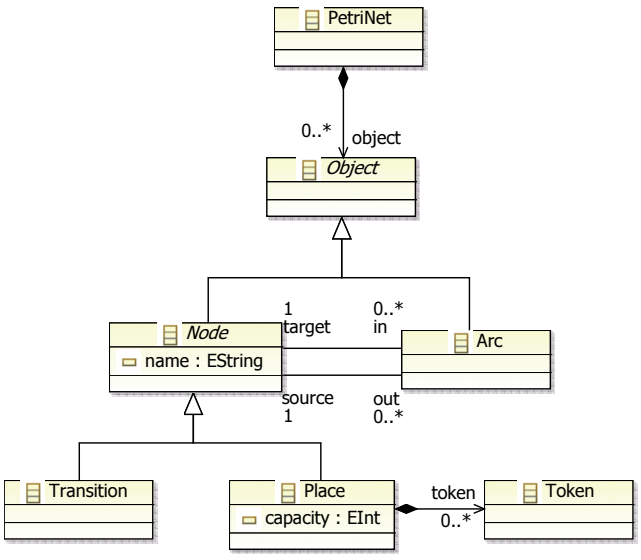

Figure 2.3: Abstract syntax of Petri nets: Ecore model

Figure 2.3 shows the Ecore model, which formalizes the concepts or the abstract syntax of Petri nets. Basically, a Petri net consists of objects, which are either nodes or arcs, where the nodes can be places or transitions. The arcs can connect nodes with each other, where each arc has exactly one

source and one target node. Each node can have a name, which is a String, and places can contain any number of tokens.

Note that, in Petri nets, there are some additional constraints, which say that an arc must not connect a transition to a transition or a place to a place directly. This can be expressed by so-called OCL constraints. We do not discuss these OCL constraints here, but you will find them in the project `APetriNetEditorIn15Minutes`, if you import it as a source project. In EMF, there are different ways of adding OCL constraints; in this project, they are added as so-called link constraints in the definition of the GMF editor[1].

Note that the model from Fig. 2.3 also has a capacity for places, which we did not discuss yet. This capacity restricts the number of tokens that are allowed to be on a place (if the capacity is a positive number). This condition can also be expressed as an OCL constraint and added to the model. We will come back to the place capacity later, when we discuss automatic roll back of interactions that violate a constrain after they are executed. For now, the place capacity can be ignored.

Together with the OCL constraints, the model from Fig. 2.3 completely and concisely define the abstract syntax of Petri nets. We discuss some subtleties of this definition later in this chapter.

2.1.3.2 Semantics of Petri nets

Next, we formalize the semantics of Petri nets in ECNO, which defines the token game or the firing rule for transitions. This semantics consists of two parts: The global behaviour is formalized by an ECNO coordination diagram, and the local behaviour is formalized by an ECNO net for each element type.

We start with the discussion of the global behaviour, with some references to the local behaviour that we discuss in more detail later. Figure 2.4 shows the coordination diagram that defines the global behaviour of Petri nets. It defines the event types fire, add, remove, and removeToken with some parameters, which we ignore for now – we come back to these parameters later. The only event that is visible at the GUI is fire.

Only the element type Transition is associated with the externally visible event fire. Therefore, we start explaining the coordination diagram there. The element type Transition is associated with three event types: fire, add, and remove. Technically, this can be seen by there boxes with the respective labels, which are called *coordination sets*. We will see later in Fig. 2.5 that the local behaviour of the element type Transition requires that all three

[1] If you have GMF installed in your Eclipse, you can have a look into the models defining the GMF editor of Petri nets. The constraint can be found in the GMF mapping file "PetriNet.gmfmap" in folder "model" of the plugin project `APetriNetEditorIn15Minutes`.

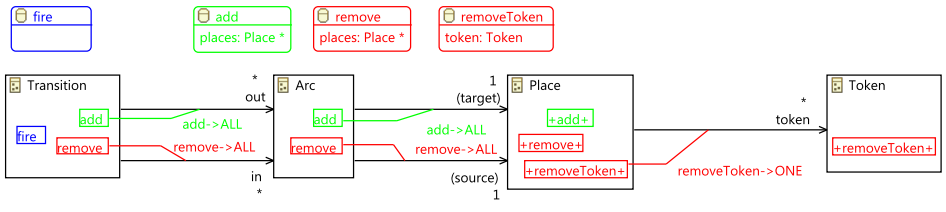

Figure 2.4: Coordination diagram: Global behaviour of Petri nets

event types fire, add, remove must be executed together. The coordination annotations attached to the coordination sets for add and remove and to the references *out* and *in* respectively, require that all the arcs starting at the Transition (*out*) need to participate in an add event, and that all the arcs ending at the Transition (*in*) need to participate in a remove event.

We will see later when discussing the local behaviour for Arc, that arcs can always participate in add and remove events. The coordination annotations of reference *target*[2] in Fig. 2.4 in addition requires that the Place to which the arc points to participates in the add event, too. Likewise, the coordination annotations of reference *source* in addition requires the Place at which the arc starts to participates in the remove event, too. This way, the coordination diagram guarantees that every place in the present of a transition is involved in a remove event, and every place in the postset of a transition is involved in an add when the transition is involved in a fire event.

The local behaviour of a Place when it participates in an add event creates and adds a token to this place. The place does not require other associated elements to participate in the add event. Therefore, the coordination set for add is not attached to any coordination annotation. Note that the label of that coordination set is enclosed between two plus signs, which indicates a subtlety that we discuss later.

The local behaviour of a Place when it participates in a remove event requires the Place to also participate in a removeToken event, which will actually take care of getting hold of a token – which then will be removed by the place. In order to get hold of one Token, the coordination annotation attached to reference *token* requires one Token to participate in the removeToken event. This way, the coordination diagram makes sure that one Token

[2]Note that the name of this reference is shown in parentheses in this diagram. The reason is that the type of the target is restricted to the class Place as compared to the class diagram of Fig. 2.3 where the target is referring to class Node. Therefore, the coordination annotation requires to propagate an add or a remove event to source resp. target elements of class Place only. In our example, this does not make any difference though. When formalizing the semantics of signal-event nets, this subtle issue is relevant – which is discussed in [34].

is involved for each place in the preset of the transition that can participate in fire event. Also for these events, we discuss why the names are enclosed between plus signs later.

Next, we discuss the ECNO nets that define the local behaviour for the different elements. The ECNO nets for these behaviours are shown in Fig. 2.5–2.8.

Figure 2.5 shows the ECNO net for the local behaviour of the Transition. There is only one transition, which does not have any places in its preset. This means that the transition can be fired any time. The event binding attached to this transition, which is indicated in bold-faced font, requires that the events fire, remove, and add must be executed together. Note that the order in this event binding does not have any meaning. The reason that the event bindings are represented as assignments is that we need a way to refer to the parameters of the events sometimes, which is through the variable the event is assigned to. We will come back to that later, when we discuss the action which is shown below the transition, which prints out the places from which the transition removes and adds tokens.

Figure 2.6 shows the ECNO net for the local behaviour of the Arc. There are two transitions, which can be executed anytime, which means that the Arc can participate in the events add and remove any time. Since the transitions are independent of each other, an arc can even participate in both events in parallel – we come back to this later. Actually, the participation of an arc in any of these events does not have any local effect on the Arc at all. The Arc element is a mediator only, which propagates the respective event from the transition to the respective places in the pre- and postset of the transition.

Figure 2.7 shows the ECNO net for the local behaviour of the Place. Again, there are two transitions in this net, which are enabled all the time and can be executed in parallel. The first transition is bound to the add event. In this event binding, we can see an example of a parameter assignment, where the resp. place assigns itself (`self()`) to the parameter *places* of the add event – we come back to this later. Moreover, the action, a Java code snippet which is executed when the add event is executed, creates a new token (using the factory that is generated by EMF from the Ecore model of Fig. 2.3). This token is then added to the place itself (`self()`) using the API generated by EMF. Note that, in order to access the factory class for creating the token, this class is imported and this factory is stored as an attribute in the two labels of that net at the top. The first label is an import label following Java syntax; the second is an attribute definition label following Java syntax. The other transition is bound to events remove and removeToken. The place assigns itself as a parameter to the remove event. In the code snippet for the action that is executed when the local behavior participates in these two elements, the token (which will assign itself to the event removeToken) is removed from the place.

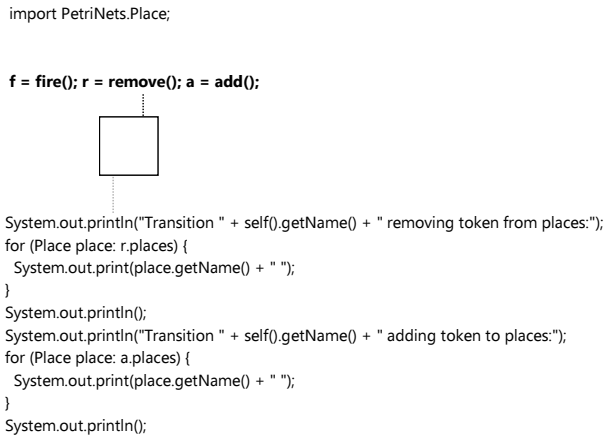

Figure 2.5: Behaviour of a Transition

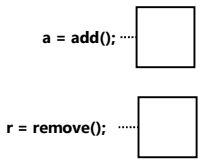

Figure 2.6: of an Arc

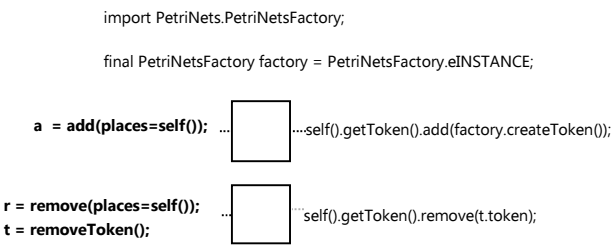

Figure 2.7: Behaviour of a Place

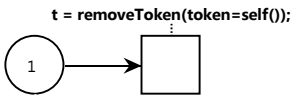

Figure 2.8: of a Token

Figure 2.8 shows the ECNO net for the local behaviour of the Token. This is the only ECNO net in this example where the firing of the transition is restricted. Actually the single token on the place makes sure that a token can be removed only once – the very semantics of a token. In the event binding, the token assigns itself to the token parameter of the event removeToken, so that the place can actually remove it. Note that in this semantics, the place is responsible for creating and removing the tokens. If the token would remove itself, the models would be slightly simpler (see [34]).

In Fig. 2.7, we can see that each place assigns itself to the remove resp. add event. For defining the semantics of Petri nets, this would note be necessary at all. We added these parameters in order to make the example a bit more interesting. In Fig. 2.4, we can see the declaration of these parameters. They are marked with an asterisk, which indicates that these are *collective parameters*, which means that every participant of the interaction can assign a value to this parameter in the event binding. And when the value is used, it represents a collection of all the assigned values. In our example, these values are used in the ECNO net for the transition (Fig. 2.5). In the action of that net, the values assigned to the add event are accessed via `a.places`; `a` is the variable to which the event add is assigned to in the event binding, and `places` is the name of the parameter – as defined in the event type definition. In the rather long code snippet, `r.places` and `a.places` are used for accessing and iterating over all places that are involved in adding or removing a token when the transition fires. Via `self()`, the action can also access other attributes of the element – in our example, this is the used to access and print out the name of the respective transition. All the rest is basic Java code. Since the Jave code snippet of that action needs to refer to the class Place, which is generated by EMF, the ECNO net needs to import this class in the import label at the top.

In Fig. 2.7 and 2.8, we have seen how an event binding can assign a value to the parameters of its events. This is done by referring to the name of the parameter and some expression. Note that that expression might even refer to other event parameters. In principle, any element participating in an event can assign a parameter to an event. When the parameter is not a collective parameter, however, the value assigned by different elements to the same parameter of the same event must be the same, that is why we call it an *exclusive parameter*. If two partners assign different values to the same parameter, the interaction will not be valid. The parameter *token* of event removeToken is such an exclusive parameter. Due to the structure of our model, however, we can be sure that there will never be more than one partner assigning a value to this parameter.

Together, the models for the local behaviour (Fig. 2.5–2.8) and for the global behaviour (Fig. 2.4) define the semantics of P/T-systems. Figure 2.9 shows an example of one interaction that is possible in that Petri net in

the given marking. The interaction is shown as an octagon containing all instances of events, the dashed lines showing the elements with which each event is associated. In order not to clutter the diagram, we have left out the parameters that are assigned to the different event instances. Note that there is only one instance of a fire, add, and remove event – since this same instance is propagated to all the elements. In contrast, there are two instances of event removeToken, which are freshly created when each of the two participating places encounters the remove event: The local behaviour of the Place requires to synchronize the remove event with a removeToken event. Executing the interaction shown in Fig. 2.9 corresponds to firing transition t1 with the top-most token on place p1.

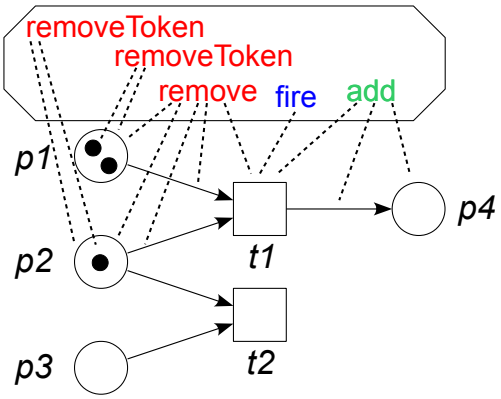

Figure 2.9: Interaction: Firing transition t1

For the situation of Fig. 2.9, there would be one other interaction, which would correspond to firing transition t1 with the other token on place p1. Beyond these two interaction, there are no other valid interactions in this situation.

2.1.3.3 Discussion

As mentioned above, the ECNO semantics from Sect. 2.1.3.2 has some subtleties, which require some discussion. To this end, let us have a look at some special situations in Petri nets: Figure 2.10 shows an interaction in a Petri net with a loop, that is where a place is connected to a transitions with two arcs in opposite directions. In that interaction, the place p1 is involved in an add event as well as in a remove event. It depends a bit on the particular kind of Petri net, whether the transition should be enabled in this situation or not: In *Elementary Net Systems* [52], for example, transitions with loops would never fire; in P/T-systems, however, the transition t1 would be able to fire. According to the ECNO semantics that we defined above, the interaction shown in Fig. 2.10 is valid. But, this is possible only

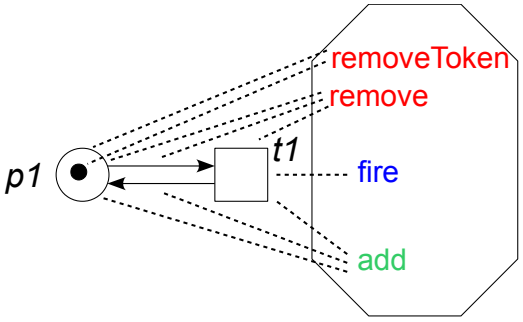

Figure 2.10: Interaction A transition with a loop

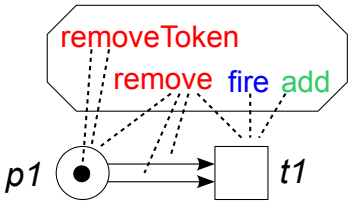

Figure 2.11: Illegal interaction: Transition with duplicate arcs

because the local behaviour formalized in ECNO nets for places allows parallel firing of transitions – if the transitions are enabled in parallel. In the ECNO net for places (see Fig. 2.7), the transition associated with the add event and the transition associated with the remove event can be fired in parallel – this way, allowing the local behaviour of the place participating in the add and remove event within the same interaction. Therefore, the above semantics formalizes the semantics of P/T-systems rather than the semantics of Elementary Net Systems.

Another situation, which requires some more discussion is shown in Fig. 2.11. In this Petri net, there are two arcs from place p1 to transition t1. The first question is, whether this is a legal Petri net; if it is legal, the second question is, what that would mean. In P/T-systems, multiple arcs between the same elements would be allowed and their meaning would be that actually each arc would require to consume one token from the respective place. In the situation shown in Fig. 2.11, however, the interaction consumes only one token. But, we will see in a minute that this interaction is actually illegal. The reason is that the remove event for place is a so-called counting event type for the place, which is indicated by enclosing the event name in the corresponding coordination set between two plus signs (see Fig. 2.4). This means that the place needs to participate in that event as many times as this is required by the arcs. In the situation of Fig. 2.11, there are two arcs starting at the place and both of these arcs participate in the remove event (indicated by the dashed lines). Therefore, the place itself

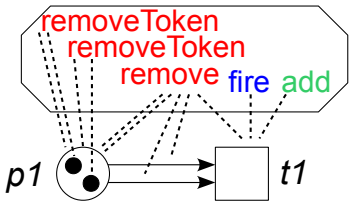

Figure 2.12: Legal interaction: Transition with duplicate arcs

would need to participate twice in the remove event. And since removeToken is a counting event type for place, it would also need to participate twice in a removeToken event. This would also require two tokens to participate in the removeToken event – or one token to participate twice in the removeToken event. The local behaviour of a token (see Fig. 2.8), however, does not allow the same token to participate twice in any event. Therefore, the interaction shown in Fig. 2.11 is invalid. But it is invalid only since the events remove and removeToken were made counting event types of elements Place and Token. If there had not been counting event types in the ECNO coordination diagram, the interaction from Fig. 2.11 would be legal – effectively ignoring duplicate arcs.

With the counting events, however, a legal interaction would require two tokens to be consumed from the place when transition is fired. This is shown in Fig. 2.12. The important thing here is that place p1 is associated with the remove event twice. And, therefore, the place needs to associated twice with some removeToken events; and due to the local behaviour of tokens, this need to be two different tokens.

For our Petri net semantics, the firing of a Petri net transition corresponds to a legal interaction in which an element of type Transition participates in an event fire. The ECNO semantics has some interactions in which no transition participates: for example, one token participating in a removeToken event only would make a legal interaction. In the ECNO, we express the interactions that are supposed to happen or are meaningful by distinguishing some elements and some events as GUI events[3]: In our example, Transition is the only GUI element type, and fire is the only GUI event. Therefore, the token cannot participate in an interaction alone – it needs to be triggered by a fire event on a Transition.

A related issue is that the local and global behaviour of ECNO define which interactions are possible in a given situation. It does not define which interactions will actually happen. This is left to external controllers; in our example, the external controller is the user who controls which interactions

[3]Note that the name and the concept of GUI events might change in future versions of ECNO.

should executed via the GUI panel. But, these controllers could also be programmed. The idea of defining what can happen, but not defining what must happen is in line with many notations for modelling concurrent or distributed systems – not least, this is in line with Petri nets. Petri nets define which transitions could fire and what happens when they fire; Petri nets do not define which transition must fire. ECNO's concept of interactions exactly reflects this idea. What must happen can be defined on top the models – in ECNO, this is done by so-called controllers (see Chapter 5 for details).

What is more, the level of atomicity of firing a transition is exactly reflected by the level of atomicity of the respective interaction representing the transition firing. This shows that ECNO allows us to capture the notion of atomicity explicitly – but, in a much more flexible and dynamic way than Petri nets.

2.2 ECNO concepts

In this section, we discuss the concepts of the ECNO modelling notation as well as their meaning in more detail. The meaning of ECNO is defined in terms of what legal interactions are in any given situation, and what happens when a legal interaction is executed in a situation.

Remember that we discuss the core concepts of ECNO here only. The discussion of the concepts of inheritance for element types as well as for event types is deferred to Chapt. 4.

2.2.1 Concepts from object-orientation

Since ECNO is based on object-oriented modelling, we briefly rephrase the concepts of object-orientation that ECNO builds on. This, basically, are a subset of UML *class diagrams* for the models, and UML *object diagrams* for the dynamic changes. Concerning terminology, we stick mostly to the terms of EMF [8] since this is the default technology for ECNO.

Note that object diagrams represent the state of a model at runtime; therefore, they are also called *runtime model* or *computation model*.

The main modelling concepts are *classes* and *references* from one class to another. The computation model defines which configurations correspond to a *class diagram* with its classes and references at runtime. This is captured by an *object diagram*, which consists of *objects* and *links* between the objects – an object diagram is also called an instance of the class diagram. Each object has a class as its *type* – and the object is called an instance of that type. Also each link from one object to another object has a *type*, which is a reference between the types of the respective objects.

In object-orientation, and specifically in UML, MOF and EMF, references have *multiplicities*, which give a lower and upper bound for how many

links of that kind of reference may exist for a specific object. The ECNO relies on the underlying semantics of multiplicities of the underlying object-oriented technology. But, it does not refer to or exploit the multiplicities explicitly; for any given object in a given object diagram and for any reference, ECNO assumes that the object has a set of links corresponding to that reference. In some cases, this set might be required to be a singleton set by the multiplicities in the object-oriented model – but ECNO deals with it as any other set in its semantics.

In class diagrams, classes can have *attributes* too. As for multiplicities, ECNO does not explicitly exploit attributes; it just assumes that the attributes are there, and that each object of some class that has an attribute has associated a value with that attribute in the respective object diagrams. Likewise, classes can have *methods*. Like attributes, ECNO does not make explicit use of these methods at all. But, the parameter assignments, conditions, and actions, in the local behaviour of elements may make use of these methods in their Java code snippets.

With our focus on class diagrams as primary object-oriented model, object-orientation is clearly biased towards structural aspects. This way, we deprive objects of a concept of behaviour in their own right. Classically, behaviour would be considered to be one of the defining characteristics of objects [6]. The reason for our non-traditional and structural presentation of object-orientation is that we want to put the limelight on the new way in which behaviour is incorporated on top of object-oriented models by solely attributing behaviour to elements and not to objects.

We assume that readers are familiar to these notions and in particular to the relation between class diagrams and object diagrams as their instances. Therefore, we do not go into more details here. For an example, you might have a look at the class diagram from Fig. 1.1 and an example of one instance in Fig. 1.2 again (see page 5 in Sect. 1.1).

Another concept which we adopt from object-orientation are *packages*, which basically provide a namespace in which "things" are defined. We assume that each "thing" that is defined in a namespace has a name and that each name is unique within the namespace. For now, we use the concept of packages very loosely. We will come back to the concepts of packages later in the context of tooling, and when splitting up models and composing models from different parts, which is more of a technical issue.

2.2.2 Events

Now, let us have a closer look at the ECNO concepts on top of object-orientation. We start with discussing the concepts of *events*. Similar to classes and their instances, we distinguish *event types* and *event instances*. The event type is what is defined in the model, the event instance is its occurrence or participation in an interaction. Similar to the relation between

objects and classes, each event instance is associated with an event type. Note that we sometimes use the term *event*, when it is clear from the context whether we refer to a type or an instance.

An event type is defined in the scope of an ECNO package, with a name that is unique in the scope of the package. An event type can have *parameters*; each parameter of an event type must have a name and a type, which refers to a class or data type from the object-oriented model. The name of each parameter must be unique within the event type.

In the computation model, the event instance might have an associated value for each parameter. In order to distinguish between the parameter of an event type and the associated value in the event instance, we call the parameter of the event type a *formal parameter*, and we call the value associated with the formal parameter in an event instance the *actual parameter* – following classical programming languages terminology [3]. Note that it is possible that some event instance does not have a value for some of its formal parameters.

Events are used to coordinate and synchronize different elements that are supposed to participate in an interaction. The event parameters are used by the different elements to share or exchange some values during the interaction in a controlled way. Typically, the actual parameter is assigned by one partner of the interaction and is used by the others (we will see in Sect. 2.2.4 how the parameters can be assigned and used). In contrast to method invocation where the caller provides the actual parameters and the callee uses them, the contribution of actual parameters and their use is completely symmetric among all participants in an interaction. In principle, any partner could contributed parameters to an event, and all the partners sharing the event can used the value. There is not caller or callee, there are only participants in an interaction – we come back to that when discussing interactions in more detail in Sect. 2.2.6. It is even possible that two partners contribute a value to the same formal parameter of an event; in that case, however, the respective interaction is legal only, if both partners contribute the same value – in the sense of the `equal()` method. At least this applies to the default kind of parameters of an event, which we call *exclusive parameters*. In our Petri net example of Sect. 2.1.3.2, the removeToken event has an exclusive parameter `token`, which is contributed by the token and used by the place to actually remove the resp. token from the place.

The other kind of parameters are called *collective parameters*. For this kind of parameter, every participant of an interaction can contribute a value to this formal parameter – but each participant can contribute at most one value. The value of the actual parameter, when accessed by a participant of the interaction, will be a collection of all the values that where contributed to it by some participant of the interaction. In our Petri net example, the parameter `places` of both, the add and the remove event, are collective parameters; they are used to collect all the places that participate in the

add event (places of the postset) and the remove (places of the preset). In the event type declaration in the coordination diagrams, the collective parameters, are indicated with an asterisk as decoration (see Fig. 2.4).

In the coordination diagrams, the declaration of event types looks very similar to the declaration of classes in class diagrams, with the parameters represented similar to attributes. But, we chose rounded rectangles for representing event types, and do not call the parameters attributes. The reason is that the role of event types and event instances is fundamentally different from the role of classes and objects (or later of element types and element instances). Event instances are volatile in nature; they live for the duration of the execution of an interaction only, which is atomic and – at least conceptually – instantaneous. Event instances do not have any meaning outside an interaction and between different interactions. On the one side, they play a similar role as actions in process algebras for synchronizing different partners; on the other side, they allow to pass parameters between different partners, somewhat similar to methods, but not quite like methods due to the symmetric nature of events as discussed above.

2.2.3 Elements

The other new concept of ECNO are elements, or more specifically *element types* and *element instances*, which we call *elements* for short. The relation between element types and element instances is similar to the relation between classes and objects.

The more interesting question is what is an element type as compared to a class? Basically, an element type extends a class by two things. First, elements have a *life cycle*, which means that element types must have a model for the life cycle or the local behaviour. Second, the life cycle of the element is defined in the way the element participates in events. To this end, each element type needs to define the event types, its life cycle is concerned in.

Section 2.2.4 discusses the concepts for defining the life cycle of elements in more detail. So, let us discuss how element types define the event types their life cycle is involved in. Actually, this is done indirectly by *coordination sets* of the element type, which refer to event types. In the example of Fig. 2.4, the coordination sets are shown as rectangles that carry a name of some event type. As their name indicates, coordination sets play a role in coordinating the participation of different elements in the respective event. This aspect will be discussed later in Sect. 2.2.5. For now, the set of all event types that occur in some coordination set of an element type defines the set of event types this element type is involved in.

One more subtle issue of an element type is the definition of which event types should be considered the *non-counting event types* and which should be considered to be the *counting event types*. If an element is required to

participate in a non-counting event, it will participate in that event exactly once, no matter how many other elements require it to participate in an event of that type. If an element is required to participate in a counting event, it will participate in that event as many times as it is required to by other elements. The details are discussed in Sect. 2.2.6 along with the definition of legal interactions.

The counting events are enclosed between two plus symbols in the respective coordination sets of the element type. Note that being a counting event or not, is a property of the element type and not of a particular coordination set. The coordination set is just where this information is shown in the graphical representation of coordination diagrams.

2.2.4 Local behaviour (life cycle)

As discussed above, the life cycle or local behaviour is one of the distinguishing features of an element. In this section, we discuss the concepts for modelling the local behaviour. We discuss the concepts based on our examples, which used ECNO nets; but, the underlying concepts are independent from Petri nets. Therefore, we explain the concepts used for modelling the local behaviour independently from Petri nets. Only in the end, we come back to ECNO nets in order to provide a concrete notation for these concepts.

Basically, the local behaviour defines, for a given object in some object diagram and a given state of the local behaviour, which *choices* are possible for that element. To this end, the local behaviour has a notion of *state*. We call an object diagram together with all the local states for each element a *situation*. And the local behaviour of an element would also define an *initial state* for each element.

For an element in a given situation, a *choice* for that element defines, which event types are involved in the choice. The choice also defines which values are assigned to the parameters of the events of the different types; these values may depend on the current state of the element, the values of its attributes and references, and even on values assigned to some of the other (exclusive) parameters of the event by other participants already. Which event types are involved in a choice and which values are assigned to their parameters is defined by an *event binding*. In addition, a choice might depend on some additional *conditions*, which may relate the values of the event parameters and the attributes and references of the element. The event binding and the condition of a choice define whether the choice is enabled or possible in a given situation. In addition, a choice defines what happens when the interaction with this participant and this choice eventually is executed, which is defined by the choice's *action*. The action might change the state of the element, the values of its attributes and references based on the local values and the parameters of the involved events. Dependent on the

changes in the object diagram and the changed state, other choices might become enabled or disabled for the element.

All this might sound a bit abstract. Therefore, we discuss how these concepts are represented in ECNO nets. In ECNO nets, the state is the current marking of the ECNO net and the initial state is the initial marking of the ECNO net. All ECNO nets of our example from Sect. 2.1.3 except for the ECNO net for Token have no places; therefore, they have a single possible state only, which never changes. The ECNO net for Token, which is shown in Fig. 2.8, has a place with one token on it, initially. So, this ECNO net has more than one state. Technically, its state space is infinite (there could be an arbitrary number of tokens on the place); but there are only two reachable states – zero or one tokens on the place. This way, it is made sure that the transition bound to removeToken can fire only once. Actually, each enabled transition of an ECNO net is a possible choice in the given state (marking). The label `t = removeToken(token=self());` attached to this transition is the *event binding*: In this example, the choice refers to one event type only: removeToken; the particular instance of this event is available in the context of this choice under the name `t`, which is expressed by the assignment. In this example, however, it is not used anywhere – but, the assignment needs to be there for syntactical reasons anyway. The event binding `t = removeToken(token=self());` also indicates how the choice contributes a value to the parameter `token` of the event type removeToken by an assignment referring to the name of the parameter: `token`. In general, there can be a comma-separated list of any number of such assignments. On the right-hand side of the assignment can be any Java expression. The expression `self()` refers to the element itself, which technically is the object underlying the element. From there, one could access all the objects (elements) attributes and references to compute some value. In principle, the expression could navigate even further, but we recommend to use "local values" of the element itself only when calculating parameter values. In addition to local values of the element itself, the expression could also access values of the event itself or (if there is more than one event in the event binding) of other events bound in this event binding. In our example, this could for example be `t.token`, which would assign the parameter `token` to itself. In this case, nothing would be assigned at all to this parameter. Sometimes, it makes sense to assign a value of one parameter to another – as long as there are no cyclic dependencies these kind of assignments will be properly resolved by the ECNO engine. In case of cycles, the respective parameters are not assigned any value. Note however, that collective parameters of events should not be used in expressions that calculate values of other parameters[4]. We will see some other examples of expressions that

[4]It is syntactically possible to use collective parameters when computing values of other parameters. This might, however, give undesired and non-deterministic effects. Therefore,

use event parameters shortly. In the example of Fig. 2.8, there is no explicit action attached to the transition. Anyway, when the choice corresponding to the transition is executed, the state of the element changes: the token is removed. This part of the action is implicit in the ECNO net. If changes in the object itself or in its links are necessary, this would need to be expressed in an explicit action attached to the transition, with a code snippet manipulating the object itself.

Fig. 2.7 shows an ECNO net with explicit actions, which change the object itself. The action of the top transition creates a new token and adds it to the place. The action of the bottom transition deletes the token, which is passed to the removeToken event as a parameter by another participant of the interaction: the token. The event is accessed by the name `t` to which it is assigned in the event binding of that transition, and the value assigned to it is accessed from there by the name of the parameter: `t.token`. This is the way how parameters can be accessed in general: in parameter assignments, in conditions, and in actions. Note that in the case of a collective parameter, the Java type of the respective expression is a Java collection over the type of the parameter.

When the Java snippets for parameter values, conditions, or actions need to refer to some classes, it is convenient to import these classes explicitly to the ECNO net by import statements. Likewise, we sometimes need to access some fixed values. To this end, ECNO nets allow us to define local attributes. Examples of this are shown in Fig. 2.5 and 2.7. Note that you should make sure that these attributes are constants only – otherwise ECNO's mechanism for saving and loading the state of an ECNO application might loose some information.

Fig. 2.5 and 2.7 show examples with event bindings with more than one event type. Generally, an event binding can have any number of events. Note that the order of the events in the event binding does not matter at all.

Note that some transitions in our examples are enabled concurrently to each other and can be fired in parallel. In our example of the Petri net semantics, this is exploited for properly capturing the semantics of loops (see discussion in Sect. 2.1.3.3). It was even necessary that the same transition could fire in parallel to itself; this was needed for properly dealing with multiple arcs between the same nodes. In general, the local behaviour of an element type might allow parallel enabledness and execution of many choices in a given situation.

---

this should be avoided.

2.2.5 Coordination

At last, we come to the core concept of ECNO: the coordination of events among different elements, which defines the global behaviour of the system. Basically, this is expressed by *coordination annotations*, which are associated with the references between element types. A coordination annotation says which other elements need to participate in an event, when an element participates in an event of some type.

Technically, the coordination is slightly more involved in order to increase the expressive power of coordination: As we have seen before, each element has *coordination sets*, each of which is associated with some event type. Note that there can be more than one coordination set for the same element type and for the same event type. Each coordination set in turn can be associated with any number (including 0) of *coordination annotations*, which refer to an event type, and are associated with a reference of the element type and have a *coordination quantifier*: ONE or ALL (for examples, see the coordination diagrams from Fig. 1.6 and Fig. 2.4).

The meaning of this is as follows: If an element el of some type t participates in an event ev of some type et, then one of the coordination sets c of the element type t for event type et is selected. For this selected coordination set all the coordination annotations associated with the coordination set c must be met for the element el and event ev. By allowing more than one coordination set for the same event type for an element, we can model a choice of different coordination annotations that need to be followed up together for that event type. This allows us to formulate disjunctions of conjunctions of coordination requirements for each event type.

We still need to define what it means that a coordination annotation is met for an element el and event ev. Let us assume that the coordination annotation is associated with some reference r and L is the set of all the links of element el with respect to this reference r in the current situation. If the coordination annotation has qualifier ONE, one link $l \in L$ must be associated with event ev, too; if the coordination annotation has qualifier ALL, all links $l \in L$ must be associated with event ev, too.

When a link l is associated with an event ev, this in turn requires that the element to which the link points must participate in the event ev, too.

Note that the set of coordination annotations associated with a coordination set may be empty (in our example, the coordination set of the event removeToken of the Token is empty). If such a coordination set is selected for an element, this means that there are no additional requirements for further elements to participate in the respective event.

In our example, we have seen that, sometimes, a reference of the class diagram is restricted to some subtype in the respective coordination diagram: For example, there is a reference `target` from Arc to Node in the class diagram from Fig. 2.3; in the coordination diagram of Fig. 2.4, the co-

ordination annotation `add->ALL` is attached to the reference *target*, but the element type it is pointing to is not Node, but one of its subclasses: Place. In that case, it is required only that all the links pointing to an object of type Place are associated with the respective event – implicitly restricting the set of links to the ones ending at an object that implements the respective subclass. In order to make the user aware of this implicit restriction, the name of such restricted references is shown in parentheses in the ECNO coordination diagram.

Note that coordination sets can be assigned a priority. If there are two or more coordination sets with the same event type for an element type, interactions including a coordination set with higher priority (for the same element) take priority over interactions in which the lower priority coordination set was selected for meeting the coordination requirements of the element.

For *non-counting event types* of an element type, there is at most one event for every type associated with each element that participates in an interaction. For *counting event types* of an element type, there can be more than one event of the same event type associated with the element. In fact, the same event can be associated with the same element more than once. The number of times an element should be associated with an event of the respective type depends on the number of incoming "triggers": each link associated with an event counts as a trigger for the element the link points to. But also the local behaviour can "trigger" some events: if based on a trigger for another event type, a choice is associated with with the element, and if this choice requires an additional event type this is counted as a "trigger", too. For example, in the local behaviour for the Place shown in Fig. 2.7, the event type remove triggers the transition with the event binding for event types remove and removeToken. Since this choice was triggered by a remove event, this is counted as an additional trigger for the other event removeToken. This way, the number of remove events and the number of removeToken events in which a Place is participating in is the same.

This idea of counting "triggers" is the reason why counting event types of an element type were called *parallel trigger event types* in some early publications on ECNO.

2.2.6 Interaction

In Sect. 2.2.5, we have discussed the main idea of the semantics of ECNO's coordination annotations. In this section, we will discuss some of the underlying concepts in more detail and make some of these runtime concepts more explicit – without repeating what was said earlier. Later in Chapter 3 we formalize a core fragment of ECNO and its semantics by formal definition. Here the focus is on the concepts of *interactions*, as what is often called a *computation model* or *runtime model* of a notation.

As mentioned earlier, in a given situation, the models for global and local behaviour of ECNO define which *interactions* are valid or enabled in that situation. First, we need to clarify what a *situation* is. It consists of all the parts, which come from object-orientation, and are captured in an object diagram, with its objects, associated values for attributes and the links between objects. In addition, a situation associates a state of the life cycle with each element (the objects which exhibit behaviour in our setting). This way, the actual state of an element consists of the values for its attributes, its links to other objects plus the state of its life cycle or local behaviour.

In such a situation, an *interaction* consists of a set of elements, which are associated with some events of some type, where the same events may be shared among different elements. And each event is associated with some values for its parameters (where some parameters might be undefined). In addition, each element is associated with one or more choices, which are enabled in parallel for that element in the given situation. Each choice is associated with the events of the respective type according to its event binding, where the events in turn must be also be associated with the respective element.

Figure 2.9 on page 37 illustrates such an interaction in our Petri net example: the association of events with elements is shown by dashed lines, the association with the choices, however, is left implicit in this figure.

The events and choices associated with the different elements must be in such a way that all the requirements of the local and global behaviour are met. This in particular means:

1. The choices associated with an element are enabled in the given situation in parallel – as defined by the local behaviour.

2. The values assigned to the parameters are compatible with the parameter assignment of the event binding of every choice.

3. The conditions of each choice evaluate to true (in the given situation and the values of the event parameters).

4. For each element, the coordination conditions as defined by the coordination sets and the coordination annotations (see Sect. 2.2.5) for each event that is associated with the element are met.

Moreover, we require that the choices associated with the interaction and its elements are minimal, in order to guarantee that no unsolicited elements or events participate in an interaction. An event is shared by two elements only, if this was explicitly required by the coordination annotations; otherwise, the events are different – even if they have the same type. Moreover, if events are shared between different elements, we require that this is actually

due to a chain of coordination annotations. The details are a bit technical and are formalized in Chapter 3.

The execution of an interaction, is quite simple: Basically, this comprises executing the action of every choice that is associated with the interaction in an arbitrary order. This could result in a non-deterministic outcome when an interaction is executed; we will discuss some guidelines later in Sect. 2.2.7 that guarantee a deterministic outcome. The execution of an interaction will typically change the situation, and thus interactions that were valid before might not be valid afterwards; and new interactions might become possible. Of course some interactions that were possible before might still be possible after. This applies to all interactions that do not have any "overlap" with the executed interaction.

Note that each situation is finite, i. e. consists of finitely many objects and elements only. This implies that there are only finitely many (even though it might be terribly many) possible candidates for interactions as long as there are no counting event types around. The argument for this is that each element can be associated with at most one event for of each type. Therefore, the ECNO engine will always terminate when computing possible interactions in a given situation, if there are no counting event types in the coordination diagram. As soon as there are counting event types for some elements, this is no longer true, since the same event type might be bound to the same element an arbitrary number of times. Therefore, for models with counting events, great care must be taken so that the computation of possible interaction will always terminate. For the coordination diagram for the semantics of Petri nets shown in Fig. 2.4, there are no termination problems, since the coordination structure is acyclic.

Conceptually, an interaction is represented as a set of elements, with each element being associated with some events and choices. In order to make it easier to formalize legal interaction, we also associate some other parts with events. For example, for an element associated with an event, we also associate the chosen coordination set with the event. And also the links that we need to follow for an element for a given coordination annotation are associated with the respective event. This helps keeping track of all requirements in the mathematical definition, but also in the understanding of interactions. Moreover, this is also the main idea of the algorithm for systematically computing all possible interactions in the ECNO engine.

2.2.7 Discussion

In this section, we have discussed all the concepts of ECNO except for inheritance. We have discussed what possible interactions are and what happens when they are executed. Some of the more subtle details will be filled in later when we formalize the semantics of ECNO.

Some of the main rationals behind the design of ECNO was that it should

be possible to fully automatically compute all possible interactions in a given situation, which is the job of the ECNO execution engine. As long as there are no counting event types, this will always succeed; in the presence of counting event types, some care needs to be taken in order to guarantee termination. One sufficient condition is that the coordination annotations are acyclic; more sophisticated criteria are still to be developed.

An other and even more important rational was that the effect of an interaction in a given situation is deterministic, which means that, when executed, the resulting situation does not depend on the order in which the different choices of the elements are executed. As long as the actions of each element type make local changes only, and as long as each element is associated with at most one choice, this is obviously true. For life cycles that allow parallel behaviour (multiple parallel choices), the modeller himself needs to take care that the order of the execution of the different actions does not make any difference in the result. For our example, this is actually true.

As discussed earlier, the semantics of ECNO defines which interactions are possible in a given situation. It does not define which interactions must be executed in that situation. This choice is left to controllers, which can (automatically) be attached to the elements of an ECNO application. In most of our technical examples, these controllers are buttons which are shown in a default GUI for each GUI element and each GUI event type. Then, it is actually the user who chooses which interaction is executed by pressing the respective buttons. For more sophisticated applications, however, such controllers can be programmed using the ECNO programming framework, which will be discussed in Chapter 5.

Together with the rational that the execution of each interaction is deterministic, the idea of controllers is that there is no non-determinism resolved in the ECNO engine itself. All the non-determinism of an ECNO application is controlled by the end-user or by some programmed controllers, which chose which of the currently enabled interactions should be executed.

An other important rational of ECNO is that its interactions are executed in a transactional way, more concretely according to the *ACID principle* [13]:

- The execution of the interaction is *atomic*. This means that either all the actions of all the choices associated with the interaction are executed or none of them are executed.

- The execution of the interaction results in a *consistent* situation again, i. e. not violating any of the conditions imposed by the class diagram (e. g. by the multiplicities of references) or some additional OCL constraints or programmed constraints. In case the execution of the interaction violates a constraint, the interaction should actually be undone or rolled back.

This consistency condition cannot be met in general. But, when the ECNO engine is run as an Eclipse application (with the ECNO: Engine registry view), interactions violating any constraints will be automatically undone[5].

- The execution of the interactions happens in *isolation*, which means it appears as if all interactions were executed in some sequence one after each other, even if they were executed concurrently. The ECNO engine achieves this by a locking mechanism that acquires locks for all elements that are involved in an interaction.

  In order to make sure that this locking mechanism guarantees isolation, it is important that all conditions and actions refer to data local to the element only. But up to now, this requirement is neither checked nor enforced by ECNO. It is a recommendation only.

- The changes made by an interaction are *durable*, which means: if the ECNO application crashes for example due to a power outage, the application can be resumed from that last successfully executed interaction on.

  The ECNO engine does not fully support *durability* yet, since this would require the integration with some database system, which is still future work. Implementing the integration with a database system is mostly a technical issue, not a conceptual one.

  The ECNO engine does support saving and loading the current situation, when it is running as an Eclipse application – as discussed in Sect. 2.1.2.

Note that as long as all interactions are triggered by the user via the GUI only, there is no concurrent execution of interactions, since everything is sequentially executed in the GUI thread. But, with some own controllers, interactions can be executed from separate threads; in that case, ECNO's concurrency control mechanisms will make sure that interactions are executed atomically, isolated, and result in a consistent situation again. We will see some examples of such controllers running in their own threads in Chapter 5.

Another rational in the design of ECNO was to foster the separation of coordination from computation. Therefore, ECNO provides a dedicated no-

[5] You might remember, that we had capacities in our Petri net model. If you now add a capacity 1 to place *pend1* and also add two tokens to *idle1*, you will realize that, when you start playing the token game as discussed in Sect. 2.1.2, you can see such a role back: If you click *fire* for transition *req1* twice in a row, you will see that the second time will have no effect – the reason is that the interaction is rolled back, since there are two tokens on place *pend1* which violates the capacity constraint for this place.

tation for coordination – which cannot be abused for defining computation[6]. The coordination notation describes which partners should participate in an interaction, so that the engine can actually do the coordination – with all the benefits discussed above: among other things, the transactional execution of interactions. This way, the programmer – or the modeller – is relieved of thinking of all the technical details that would be necessary when programming the coordination in a computation oriented language. In ECNO, clearly, coordination diagrams are the major modelling notation for describing coordination; but also the local behaviour of an element is mostly about coordination. Only the actions exhibit a clear computational characteristics. We believe that this enforced separation of "stuff that is related to coordination" and "stuff that is related to computation" is a major contribution of ECNO. Events seem to be the key to describing coordination, and the proponents of process algebras [21, 4, 41] – who used the term action for what we call event – could say "told you so". The use of events for coordination is not new; what might be new in ECNO is, that a single interaction can involve many different events of different types and a complex network of involved elements – without losing the principle of interactions being atomic. Therefore, we give events the same status as objects even though – or actually because – they are fundamentally different from objects. Events live for an instant (the duration of the execution of an interaction) only.

We have seen above that coordinational behaviour can refer to computational behaviour by code snippets in the actions. Actually, it is also possible to refer to coordinational behaviour from computational behaviour. This is actually used to realize the controllers for interactions. But, we consider this mostly a technical issue which allows us running interactions within traditional computation oriented setting; therefore, we defer a more detailed discussed of this aspect to Chapter 5.

Another important contribution of ECNO is the clear separation of the life cycle of an individual object or element and the global behaviour – the global behaviour is coordination only, the local behaviour is a combination of coordination and computation. Actions, basically, represent the computation part, and coordination is the mechanism for integrating the global behaviour with the local behaviour – and, via the actions, coordination is integrated with computation.

The actions of the local behaviour allow us also to integrate ECNO with traditional object-oriented software. Moreover, the ECNO engine is fully aware of changes that are made by some classical object-oriented parts of the software, which might not even be aware of ECNO running on top of it. The ECNO engine will properly update possible interactions at any time –

[6] Actually, we believe that, in most languages used in software development today, it is the other way round: Computation-oriented languages – sometimes equipped with some additional coordination-oriented features – are used for programming coordination.

independently of whether the changes come from ECNO or other parts of the software. This way, purely object-oriented software can be step by step extended with parts that are modelled in ECNO. We discuss some of the underlying concepts in more detail in Chapter 5.

Chapter 3

Formal Semantics

In Chapter 2, we have discussed all concepts of ECNO except for inheritance, and we have explained their meaning, in terms of valid interactions and their execution. In this chapter, formalize this semantics. For the time being, we formalize the semantics of a core fragment of ECNO with the focus on the coordination. One reason for restricting the formalisation to the core fragment of ECNO are time-limitations. Another reason is that restricting to the core fragment helps us focusing on the essence of coordination – not cluttering the semantics with all kinds of less important details and losing track of some important concepts in the technical details of less important concepts.

In a future version of this technical report, we might eventually add a section with the full semantic of ECNO. Actually, one idea is to formalize the full version of ECNO using the core fragment of ECNO.

3.1 Basic definitions

In this section, we introduce the basic notations from mathematics that we need for the formalisation and introduce a simple definition of the concepts of class diagrams and their instances, object diagrams which are underlying the concepts of ECNO. But, we keep these definitions as simple as possible – they are not meant as a formalisation of UML in any form.

3.1.1 Basic notation

Throughout the formalization, we use the usual notations from set theory. We use $\emptyset$ to denote the *empty set*. For a given set A, we denote the set of all subsets of A by 2^A, which is called the *power set* of A.

A mapping $f : A \to B$ is *injective*, if for all elements $x, y \in A$ with $x \neq y$ also $f(x) \neq f(y)$. For some mapping $f : A \to B$ and some subset $A' \subseteq A$, we denote the mapping f restricted to the elements of A' by $f|_{A'}$.

3.1.2 Class diagrams and object diagrams

In this section, we formalize a simple version of class diagrams and object diagrams. We keep this formalization as simple as possible, but powerful enough to capture the important features of ECNO. Note that we do not formalize attributes, nor do we formalize multiplicities, since they are not relevant for ECNO – we assume that all references are of multiplicity many. The underlying object-oriented technology will take care of modelling and maintaining the correct multiplicities in the underlying object diagrams; ECNO just builds on it.

Definition 3.1 (Class diagrams) *A* class diagram $\mathcal{C} = (C, R, s, t)$ *consists of two disjoint finite sets C and R and two mappings $s : R \to C$ and $t : R \to C$.*

The elements of C are called the classes *of $\mathcal{C}$ and the elements of R are called the* references *of $\mathcal{C}$. For a reference $r \in R$, $s(r)$ is called the* source class *of the reference and $t(r)$ is called the* target class *of the reference.*

For formalizing object diagrams and, in particular, for making it easy to define changes of an object diagram, we define a universe of possible objects for each class of the class diagram; also for distinguishing different links between object, we define a universe of link identifiers for each reference. To this end, we define a universe U, which is indexed over $C \cup R$, i. e. $U = (U_i)_{i \in C \cup R}$, where all U_i are countably infinite and pairwise disjoint. For a class $c \in C$, the set U_c represents all the possible candidates of objects of type c. The universe of all possible objects is denoted by $U_C = \bigcup_{c \in C} U_c$. As a shorthand notation, we define the mapping $c : U_C \to C$ by $c(o) = c$ if $o \in U_c$. This way, $c(o)$ denotes the class of object o.

A possible link can be defined as a triple $l = (o_1, u, o_2)$, where $u \in U_r$ for some reference $r \in R$ with $o_1 \in U_{s(r)}$ and $o_2 \in U_{t(r)}$. Object o_1 is the source object and o_2 is the target object of the link, and the condition makes sure that the type of the source and target object are the type of the source and target class of the reference, respectively. For the link l, we denote the underlying reference r, the links type, by $r(l)$.

Note that we cannot represent links as pairs of a source and target object only, since there can be many different links between the same objects. To distinguish different links we use the universe as a third – the middle – component. Since we needed to introduce these unique identifiers anyway, we sneak in the type of the links to these identifiers in a similar way to the universe of objects for each class. This way some of the definitions (in particular Def. 3.4 and Def. 3.7) become much simple, which helps us focusing on the formalization of the coordination concepts.

The set of all possible links with respect to a class diagram and the given universe of objects can then be defines as $U_L = \bigcup_{r \in R} U_{s(r)} \times U_r \times U_{t(r)}$.

With these notions, we can now define *object diagram* as an instance of a class diagram. We introduce a notion of states for objects already for object diagrams. This caters for possible values of its attributes (which are not formalized in our definition of class diagrams), but also for the state of the local behaviours. This way, we technically do not need to make a distinction between object diagrams and *situations*, which conceptually are different. In order to associate states with objects, we introduce a countably infinite set S of possible states, which represent the possible *states* for each object in an object diagram or situation.

Technically, the possible states are globally defined and are the same for all objects. Therefore, all objects and elements have the same state space, technically. Since S is sufficiently large (countably infinite), the states space of different element types can be made disjoint. Therefore, the global definition of states is no conceptual restriction.

Definition 3.2 (Object diagrams) *Let $\mathcal{C} = (C, R, s, t)$ be a class diagram. An* object diagram *$\mathcal{O} = (O, L, \sigma)$ for class diagram $\mathcal{C}$ consists of a finite set $O \subseteq U_C$ of* objects*, a finite set $L \subseteq U_L \cap (O \times U \times O)$ of* links*, and a mapping $\sigma : O \to S$, which defines the* state *$\sigma(o)$ of each object o in that situation.*

Note that the type (its class) of each object $o \in O$ is indirectly defined since the object is coming from the universe of objects; as defined earlier, this class is denoted by $c(o)$. Likewise, the reference associated with a link $l \in L$ was defined as $r(l)$ for the universe of links. By definition, each object diagram is finite, even though the universe of possible objects and links is infinite. But object diagrams can be arbitrarily large.

3.2 Formalization of the core fragment of ECNO

In this section, we formalize the core fragment of ECNO, which concerns the core modelling concepts of ECNO as well as their semantics. The core fragment that we formalize here does not include

- coordination sets, (resp. we assume that there is exactly one coordination set for every event type of an element type),
- event parameters,
- counting event types,
- parallel choices (i. e. each element participating in an interaction is associated with exactly one choice), and
- inheritance.

3.2.1 Formalisation of modelling concepts (syntax)

We start with formalizing the core modelling concepts of coordination diagrams. In the core fragment, a coordination diagram defines a set of *event types* and adds *coordination annotations* to some references of the class diagram. Implicitly this means that there is exactly one coordination set (which we called default coordination set in some earlier versions of ECNO). But, if there is only one coordination set, we do not need to formalize it – resulting in a simpler definition.

We call a coordination diagram using the core modelling concepts only a *simple coordination diagram.*

Definition 3.3 (Simple coordination diagram) *Let $\mathcal{C} = (C, R, s, t)$ be a class diagram. A* simple coordination diagram *$\mathcal{C}o = (Et, A)$ based on $\mathcal{C}$ consists of a finite set Et of* event types *and a finite set $A \subseteq Et \times R \times M$ of* coordination annotations*, where $M = \{ONE, ALL\}$.*

For a coordination annotation $(et, r, m) \in A$, *et* refers to the event type, r refers to the reference it is associated with, and m is its *coordination quantifier.*

The *local behaviour* defines which choices are possible for a given object in a given situation. And for a given choice, the local behaviour defines which event types participate in it and how the execution of the choice eventually changes the situation locally. The changes are described by defining a new state for the object, and by defining which links that are deleted and which links are created for the object, when the choice is executed in a given situation.

In the definition of the behavior, the choice itself is a unique identifier only. For that reason, we can define a universe of possible choices as a countably infinite set X globally again. Since each local behaviour can use it as it pleases, this global definition of possible choices does not impose any conceptual restrictions.

Definition 3.4 (Local behaviour) *Let $\mathcal{C}o = (Et, A)$ be a simple coordination diagram over a class diagram $\mathcal{C} = (C, R, s, t)$.*

A local behaviour $B = (\iota, \gamma, \varepsilon, \delta, \alpha, \tau)$ for a class $c \in C$ consists of an initial state $\iota \in S$, and of mappings γ, ε, δ, α, and τ such that the following conditions are met: for a given object diagram $\mathcal{O} = (O, \sigma, L)$ and any object $o \in O$ with $c(o) = c$, $\gamma(\mathcal{O}, o)$ is a finite subset of X, which is the set of possible choices *for o in this situation; for each $x \in \gamma(\mathcal{O}, o)$, $\varepsilon(x)$ is a finite subset of Et, which is the set of event types the choice x is participating in; and $\delta(\mathcal{O}, o, x) \in 2^{(\{o\} \times U \times O) \cap L}$ is the* set of links that are deleted *when the choice is executed, $\alpha(\mathcal{O}, o, x) \in 2^{(\{o\} \times U \times U) \cap U_L}$ is a finite* set of links added *when the choice is executed, and $\tau(\mathcal{O}, o, x) \in S$ is the* new state *of the object.*

The local behaviour for a coordination diagram is defined as an C-indexed family of behaviors $(B_c)_{c\in C}$. We denote the behaviour for class c also by $B_c = (\iota_c, \gamma_c, \varepsilon_c, \delta_c, \alpha_c, \tau_c)$.

In the above formalisation, ι defines the initial state of the object; γ defines, for a given situation, the possible choices for an object, based on the context of the object (this is why γ takes the complete object diagram as a parameter, actually γ should exploit the links local to the object only); ε defines which event types are involved in a given choice; δ and α define which links (local to the object) are deleted and created; finally, τ describes how the state of the object changes.

Note that there is no way of deleting objects, which is in line with Java's semantics were inaccessible objects will eventually be garbage collected. But, in our formalisation below, we do not formalize a garbage collection mechanism. Note that there is no explicit notion in the behaviour for creating new objects; but if a link to an object that did not exists before is created, this indirectly creates a new object.

3.2.2 Formalisation of meaning (semantics)

In this section, we formalize which interactions are possible in a given situation and how the execution of an interaction in a given situation changes the situation. As mentioned earlier, situations are represented by object diagrams in our formalisation (exploiting the fact that objects have states in our formalization). In order to formalize interactions, we proceed in two steps. The first step is formalizing the general structure of an interaction, basically binding the participating elements to some events and choices – we call this an *interaction structure*; in a second step, we formalize, which of these general interaction structures are actually *interactions* in a given situation, which we sometimes also call *valid interactions.*

In our formal definition of interaction structures, we add already some basic local soundness conditions. The formalization of the actual coordination constraints, is deferred to a the definition of interactions (Def. 3.6). We give the formal definition of interaction structures first, and discuss its different parts in more detail right after the definition.

Definition 3.5 (Interaction structure) *Let $\mathcal{C}o = (Et, A)$ be a simple coordination diagram and $(B_c)_{c\in C}$ the local behaviours over a class diagram $\mathcal{C} = (C, R, s, t)$, and let $\mathcal{O} = (O, \sigma, L)$ be an object diagram for class diagram $\mathcal{C}$ (which is a situation for the coordination diagram $\mathcal{C}o$.*

An interaction structure *$\mathcal{S} = (O', L', E, et, v, \chi)$ for object diagram $\mathcal{O}$ consists of a set $O' \subseteq O$, a set $L' \subseteq L$, a finite set E of eventd, a mapping $et : E \to Et$, and a mapping $v : O' \cup L' \to 2^E$ such that the following conditions are met:*

1. $E = \bigcup_{o' \in O'} v(o')$;
2. *for each* $x \in O' \cup L'$, *the restriction* $et|_{v(x)}$ *is injective and* $v(x) \neq \emptyset$;
3. *for each link* $l = (o', u, o'') \in L'$ *and each* $e \in v(l)$, *there exists an* $m \in \{ONE, ALL\}$ *such that* $(et(e), r(l), m) \in A$;
4. *for each element* $l = (o', u, o'') \in L'$, *we have* $o' \in O'$ *and* $o'' \in O'$, *and* $v(l) \subseteq v(o')$ *and* $v(l) \subseteq v(o'')$.
5. χ *is a mapping* $\chi : O' \to X$, *such that for each* $o' \in O'$, $(\chi(o') \in \gamma_{c(o')}(\mathcal{O}, o')$ *and* $\varepsilon_{c(o')}(\chi(o')) = v(o')$.

In the above definition of interaction structure $\mathcal{S} = (O', L', E, et, v, \chi)$, the sets O' represents the set of objects involved in the interaction; the set L' represents the set of links that are involved in the interaction, by propagating some events between two object according to some coordination annotation. The set E is the set of all the events (event instances to be precise) that are involved in the interaction, and, for each event $e \in E$, $et(e)$ is the event type of the event e. The mapping v says which events are associated with each element and each link in the interaction structure. The conditions of Def. 3.5 mean the following:

1. Guarantees that each event of the interaction structure is bound at least to one object – events not bound to anything are meaningless in an interaction.
2. Guarantees that all events bound to the same object or link have different types (remember that, in the core fragment of ECNO, we do not cover counting event types or parallel choices; therefore we do not need different events of the same type being associated with the same object or link); in addition, we require that all objects and links that are part of an interaction structure are actually bound to an event. Objects not involved in any event, cannot be part of an interaction.
3. Guarantees that a link is bound to an event only if there is a coordination annotation in the coordination diagram, which might require such an event being bound to the link. Note that this definition does not yet take into account the quantifiers of the coordination annotations. This is covered in the definition of valid interactions.
4. Guarantees that for a link that is part of the interaction structure, the source and target object are part of the interaction structure too, and the events the link is bound to are also bound to the source and the target object.

5. Guarantees that the choice associated with an element is actually enabled in the given situation, and that the event types required by that choice are identical to the set of event types required by that choice.

These requirements do not yet guarantee that an interaction structure is a valid interaction. They are just some basic structurally necessary conditions.

We will formalize the constraints concerning the coordination annotations below. In addition, we need to formalize that all objects involved in an interaction somehow are connected, and that elements that are involved in the same event are actually connected by some links associated with the same event – in order not to "accidentally" identify two different events which just happen to have the same type. In order to formalize these and some other constraints on valid interactions, we need to introduce some additional concepts and notations.

The first notation denotes the set of objects that are reachable from an object o within an interaction structure, which is denoted by $R(o)$ – assuming that the interaction structure is fixed. For an interaction structure $\mathcal{S} = (O', L', E, et, v, \chi)$ and some object $o \in O'$, we define the set of objects reachable from o in $\mathcal{S}$ as the smallest set $R(o)$ such that

- $o \in R(o)$
- if $o' \in R(o)$ and $(o', u, o'') \in L'$, then $o'' \in R(o)$.

The second notation denotes the set of objects that are reachable from an object o by navigating only those links of the interaction structure that relate to a specific event $e \in E$, which we denote by $R(o, e)$. This will help us characterizing which events are actually allowed to be equal. For an interaction structure $\mathcal{S} = (O', L', E, et, v, \chi)$, some object $o \in O'$ and some event $e \in E$, we define the objects that are connected with respect to event e as the smallest set $R(o, e)$ such that

- if $e \in v(o)$ then $o \in R(o, e)$
- if there exists some link $l = (o', u, o'') \in L'$ with $e \in v(l)$ and $o' \in R(o, e)$ or $o'' \in R(o, e)$ then we have $o', o'' \in R(o, e)$.

A third notation denotes for an object $o \in O$ and a coordination annotation a attached to a reference of that object the set of all links $l(o, a)$ that refer to this reference and are associated with an event of the respective type. More formally, for a given object $o \in O$ and a coordination annotation $a = (et, r, m)$ with $s(r) = c(o)$, we define $l(o, a) = \{(o, u, o') \in L' \mid u \in U_r \wedge \exists e \in v(o, u, o') : et(e) = et\}$.

At last, we define, for a given object $o \in O$ and some reference $r \in R$, the set of all links with respect to reference r that start at object o, which we denote by $\hat{l}(o, r)$. More formally, we define $\hat{l}(o, r) = \{(o, u, o') \in L \mid u \in U_r\}$.

With these definitions and notations, we can now define what a (valid) interaction in a given situation is.

Definition 3.6 (Interaction) *Let $\mathcal{C}o = (Et, A)$ be a simple coordination diagram and $(B_c)_{c \in C}$ the local behaviours over class diagram $\mathcal{C} = (C, R, s, t)$, let $\mathcal{O} = (O, \sigma, L)$ be an object diagram for class diagram $\mathcal{C}$ and let $\mathcal{S} = (O', L', E, et, v, \chi)$ be an interaction structure.*

The interaction structure $\mathcal{S}$ is an interaction *of the simple coordination diagram $\mathcal{C}o$ and the behaviours $(B_c)_{c \in C}$ for an element $o \in O'$ in the object diagram $\mathcal{O}$, if the following conditions are met:*

1. *All objects in $\mathcal{S}$ are reachable from o, i. e. $R(o) = O'$.*
2. *If, for any two objects $o', o'' \in O'$, there exists an event $e \in E$ with $e \in v(o')$ and $e \in v(o'')$, then we have $R(o', e) = R(o'', e)$.*
3. *For each object $o' \in O'$, each event $e \in v(o')$, and each coordination annotation $a = (et(e), r, m)$ with $s(r) = c(o')$, we have*
   - *$l(o', a) = \hat{l}(o', r)$ if $m = ALL$ and*
   - *$|l(o', a)| = 1$ if $m = ONE$*

Let us briefly explain the conditions in the above definition:

1. Guarantees that all objects in the interaction can be reached transitively by some links of the interaction, which makes sure that objects that are part of the interaction are actually required to be there.
2. Guarantees that two objects that are associated with the same event are actually required to share the same event by a chain of coordination requirements.
3. Guarantees, for each event associated with an object, that each coordination annotation for that object and the respective event type is met. This definition distinguishes the two different cases for coordination quantifiers ALL and ONE.

At last, we need to define how an interaction is executed and, in particular, how its execution changes the object diagram. Basically, this means applying all the changes made by the local behaviours of all participating objects. Since the local behaviour is defined in such a way that the different local behaviours cannot interfere with each other, the definition is quite straight forward.

Definition 3.7 (Execution of an interaction) *Let $\mathcal{C}o = (Et, A)$ be a simple coordination diagram and $(B_c)_{c \in C}$ the local behaviours over $\mathcal{C} = (C, R, s, t)$, let $\mathcal{O} = (O, \sigma, L)$ be an object diagram for class diagram $\mathcal{C}$ and let $\mathcal{I} = (O', L', E, et, v, \chi)$ be an interaction for some object $o \in O'$.*

The interaction can be executed *in $\mathcal{O}$, which will result in a new object diagram defined by $\mathcal{O}_2 = (O_2, \sigma_2, L_2)$, where*

1. $O_2 = O \cup \{o_2 \mid (o_1, u, o_2) \in \alpha_{c(o_1)}(\mathcal{O}, o_1, \chi(o_1)), o_1 \in O'\}$
2. $L_2 = (L_1 \setminus \bigcup_{o' \in O'} \delta_{c(o')}(\mathcal{O}, o', \chi(o'))) \cup \bigcup_{o' \in O'} \alpha_{c(o')}(\mathcal{O}, o', \chi(o'))$
3. $\sigma_2(o') = \tau_{c(o')}(\mathcal{O}, o', \chi(o'))$ *for each* $o' \in O'$, $\sigma_2(o') = \sigma(o')$ *for each* $o' \in O \setminus O'$, *and* $\sigma_2(o') = \iota_c(o')$ *for each* $o' \notin O$.

If an object diagram $\mathcal{O}_2$ is the result of executing an interaction $\mathcal{I}$ in an object diagram $\mathcal{O}$, we denote that by $\mathcal{O} \xrightarrow{\mathcal{I}} \mathcal{O}_2$.

Let us briefly discuss the three points in the above definition:

1. Adds the new elements to the object diagram, which implicitly come in by the added links (α) of the local behaviours of the different elements.
2. Removes the links that are supposed to be deleted (δ) by the different local behaviours, and adds the links that are supposed to be added (α) by the local behaviours of the different elements.
3. Sets the new state for each element that is involved in the interaction according to the local behaviours (τ) of the element. For an element o' that is part of the interaction, $\tau_{c(o')}$ denotes the new state of object o'. For each element that is not part of the interaction, the state remains unchanged. For each element that is added by some local behaviour, the state is set to the initial state according to the local behaviour (ι) of the respective class.

Note that, due to our "trick" with the universes for objects of certain classes and for identifiers for links, we do not need to take care of the types of the added objects and the added links. They are implicitly contained in the objects and links themselves and this way in the δ and α of the respective local behaviours.

3.3 Summary

In this section, we have presented a first formalisation of the syntax and semantics of the core fragment of ECNO. The core fragment covers the essence of the coordinations and their meaning. In particular, the formalisation makes explicit some subtleties of valid interaction: for example, elements can only share an event, if there is a chain of coordination annotations between these elements; and there must be one root element from which all coordination originates.

Up to now, we did not formalize some of the more advanced concepts of ECNO such as coordination sets, counting events, and event parameters. But, the definition of the core fragment, might give an idea already how they could be formalized.

Ultimately, the formal definition of a semantics of a notation is of not much use – unless we use it for some purpose. Eventually, we could exploit the formal definition of the semantics for verification purposes. But, this is left for future research.

Chapter 4

Inheritance

In the previous chapters, we have discussed all concepts of ECNO except for the ones related to inheritance. We had deferred the discussion of inheritance since, at its core, coordination is independent of inheritance and the concepts of coordination are easier to understand when inheritance is not mixed in. Moreover, the addition of inheritance introduces many subtle issues and choices in the design of the ECNO modelling notation, which need to be carefully discussed and explained. Therefore, we dedicate a complete chapter to the discussion of the concepts of inheritance in ECNO.

One source of "subtle issues and design choices" is that there are many different forms of inheritance on behaviour life cycles [55]. Moreover, there is not only inheritance on classes or element types; we also need to consider inheritance on event types. And it turns out, that we need to distinguish between two different kinds of inheritance for event types, which we call *specialization* and *extension* (see Sect. 4.2.2).

We start with introducing yet another example, which exploits some forms of inheritance, and we use these examples to informally discuss inheritance in ECNO. Later, in Sect. 4.2, we give a more precise account of the different forms of inheritance and their meaning and discuss some additional constructs.

4.1 Example: Vending machine

Our example is the model of a vending machine. The example that we present here has evolved from an example that we had used for discussing an earlier version of ECNO [32, 33]; at that time, however, ECNO supported a limited form of inheritance. On the side, this example exhibits another feature of ECNO, which is important for its practical use: *packages* for structuring ECNO models and building ECNO applications from separate components. Note that we do not discuss the technical details concerning packages in this chapter. We just discuss packages as far as we

need to understand the example. For details on ECNO packages, we refer to Chapter 5.

If you want have a look at the models in Eclipse, or if you want to run the example, you need to import the following projects as source projects to your workspace:

- `dk.dtu.imm.se.ecno.examples.vendingmachine.split.part1`
- `dk.dtu.imm.se.ecno.examples.vendingmachine.split.part2`
- `dk.dtu.imm.se.ecno.examples.vendingmachine.split.part3`
- `dk.dtu.imm.se.ecno.examples.vendingmachine.split.part3.runtime`

The first three projects contain the different parts of the ECNO models; the basic model from `part1` is extended in two steps: `part2` and `part3`. The last project (with extension `runtime`) contains an instance of a vending machine, which can be run as an ECNO application. It can be started by right-clicking on the file "initial.behaviourstates" in folder "run" and then selecting "ECNO → Start ECNO Engine". Remember that the running engine can be controlled and finished via the "ECNO: Engine registry" view.

4.1.1 Structural model

We start with explaining the structural model of the vending machine. In the actual projects, the structural models are split up into three parts, too. Since the structural model is not very big or complicated, however, we explain the full model right away. This model is shown in Fig. 4.1. A VendingMachine consists of different VendingMachineComponents. Note that, for technical reasons, we made the Coins, which are inserted to the vending machine also a VendingMachineComponent, though this conceptually is not true. We explain the other real components next.

In the vending machine, there is a Panel, which serves as the front-end to the customer on which the customer can press some buttons for interacting with the machine. The Panel is attached to one ore more Controls. A Control in turn is attached to one or more Slots, which hold and take control over the Coins. And a Slot is attached to a Safe, where the coins will be passed to when a beverage is dispensed. There are two references between Coin and Slot. The reference slot from Coin to Slot represents the coins that can be inserted to the slot by the customer. The reference from Slot to Coin represents the coins that currently are inserted to the slot.

The Control is also attached to some Brewer, and there are two different kinds of brewers, a CoffeeBrewer and a TeaBrewer. There is an Output, which represents the place where the customer places a cup, to which the beverage is dispensed.

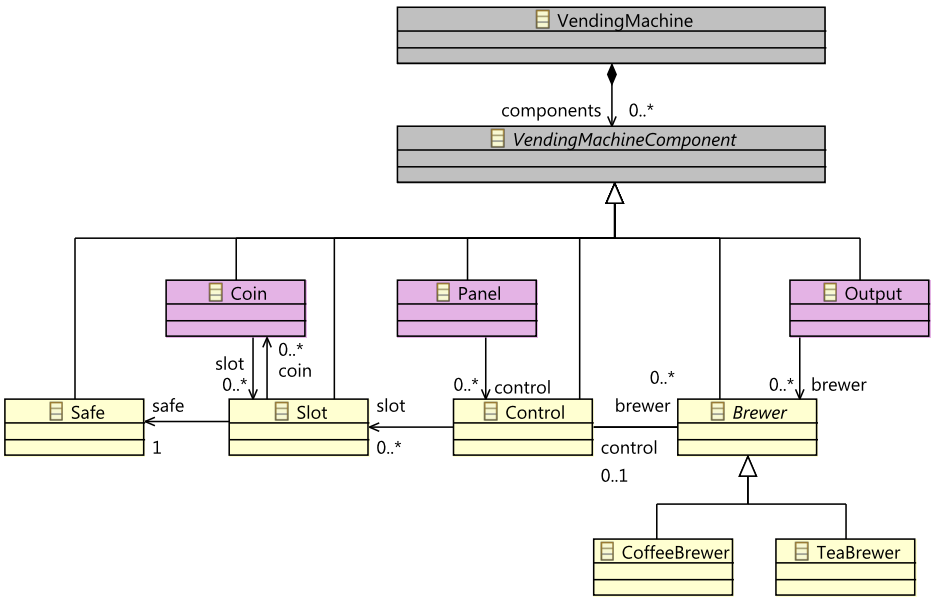

Figure 4.1: Vending machine: Structural model (complete)

Before we continue discussing some more details of Fig. 4.1, let us have a brief look at an instance or configuration of a vending machine. Figure 4.2 shows an instance of a vending machine with two coffee brewers and one tea brewer, and with three coins that are ready to be inserted to the vending machine. Note that the container object, an object of class VendingMachine, containing all the objects is omitted from this object diagram in order not to clutter the diagram.

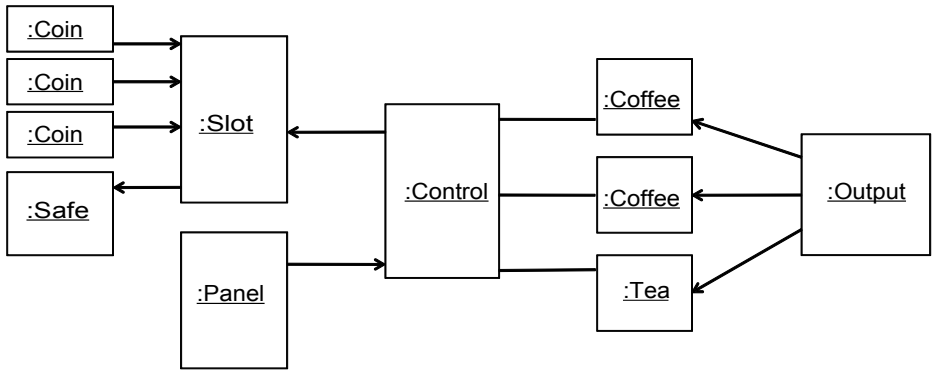

Figure 4.2: Vending machine: Instance

Note that, in the structural model of Fig. 4.1, different classes have differ-

ent colours: the two classes VendingMachine and VendingMachineComponent at the top are mostly of technical nature, which allows all components of a vending machine being contained somewhere. We will not associate any behaviour with them later in our ECNO models. The three classes Coin, Panel, and Output shown in magenta, represent GUI elements in the ECNO model. The other classes are internal.

Experienced modellers might also wonder why the class Brewer is not made an abstract class (i. e. its name shown in italics in the diagram). This is where we come back to the point that, in our projects, the models are split up into different parts. The basic version (part1) contains all the classes except for the CoffeeBrewer, the TeaBrewer and the Output. In order to allow us to play around with this basic version, we made Brewer a concrete class, so that we could have an instance of a brewer. But, this is just to demonstrate that, at all stages of development, our models are executable. A vending machine without a brewer would not be able to do anything reasonable. In the second version (part2) the Output is added to the model. In the full version (part3), all components shown in Fig. 4.1 are there, and this is where inheritance comes in, since the behaviour of the brewers consists of the general behaviour of the brewer plus the specific behaviour for coffee and for tea brewers. We will come back to that later.

Note that there is also an inheritance relation between VendingMachine-Component, which actually is abstract, and all the concrete classes for the components. This inheritance however is structural only since VendingMachineComponent as well as VendingMachine are classes only; they are not element types in ECNO (therefore, they are shown in grey in this diagram).

4.1.2 ECNO models: Part 1

Next, we discuss part 1 of the ECNO models, which captures the basic part of the vending machine. Figure 4.3 shows the coordination diagram for the elements of part 1 including the main event types. The main event types are insert, pass, return_[1], reset, cancel, drink, and cup_in. Note that there are also some event types with "odd" names, yyy, bla, and blubs, which we skip for now; we come back to them later.

Some of the event types have parameters with types that are either basic data types like String[2] or classes from the Ecore model. We explain the purpose of the different parameters later, when we discuss the local behaviour of the different types of elements.

First, let us have a look at the global behaviour which is represented by the coordination annotations shown in Fig. 4.3. We start with the events, which can be triggered by the user via the Panel. The panel is involved in

[1] Since **return** is a keyword of our target language Java, we added an underscore to the name of this event in order to avoid syntax errors in the generated code.

[2] EString is the Ecore name for Java Strings.

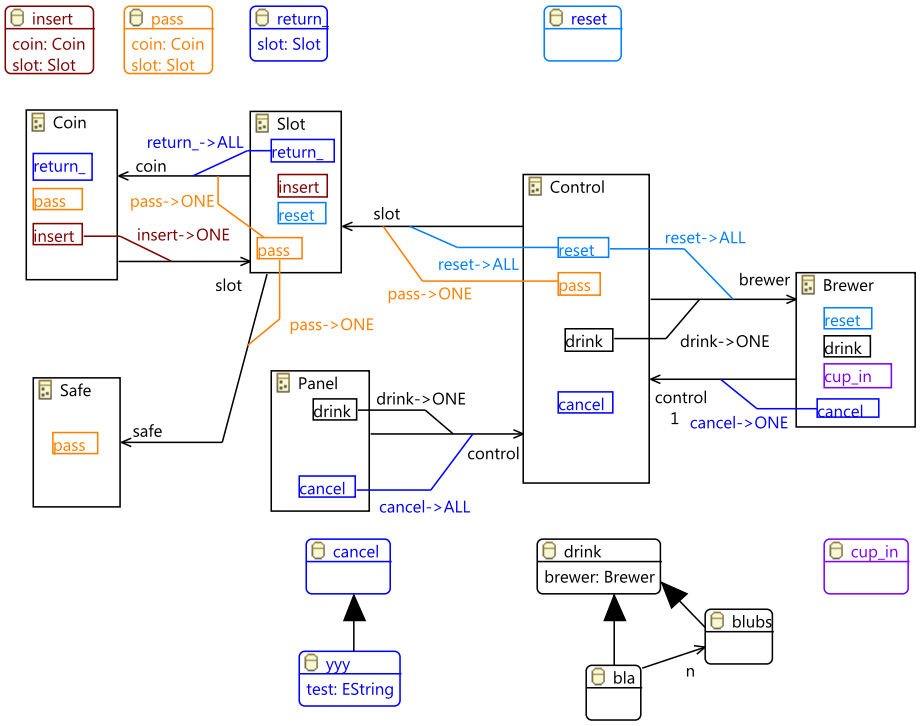

Figure 4.3: Vending machine, part1: Coordination

two types of events: drink and cancel. The drink event represents ordering and brewing a drink – and the global behaviour will make sure that the drink is paid for. The cancel event is for returning all inserted coins and resetting all internal activities of the brewers. We start with explaining the interactions that are triggered by a drink event: If a panel participates in a drink event, the coordination annotation associated with the reference to control requires that one control element participates in the drink event, too. In turn, this control requires that one brewer participates in the drink event – which will initiate the brewing of the drink in the respective brewer. There is no additional requirement for the brewer concerning the drink event. We will see later in the local behaviour of the Control that the control can participate in a drink event only together with participating in a pass event. This pass event represents the slot passing an inserted coin to the safe – representing the payment of the drink. To achieve this, the Control requires that one Slot participates in the pass event, and the coordination requirements for the Slot make sure that exactly one Coin and one Safe participate in the pass event, the local behaviour of which will actually transfer the inserted coin from the slot to the safe. Altogether, if a panel participates in a drink event – i. e. when the user presses the drink button on the panel – one brewer will take care of brewing the drink and one coin will be passed from the slot to the safe – all in the same interaction.

Similarly, the user can issue a cancel event at the panel. If the Panel participates in a cancel event, the coordination annotation to Control requires that all controls participate in the cancel event. The local behaviour of the control will make sure that the cancel event is possible only together with a reset event. This, in turn, requires that all brewers and all slots participate in the reset event. The idea is that the reset event will clean the brewers and the slot returns all inserted coins to the user. To achieve this, the local behaviour of the Slot requires that a reset event is executed together with a return_ event, which requires all inserted coins to participate and return themselves to the user. We will discuss the local behaviour in more detail later.

A last event that can be issued by a user is insert, which represents a user inserting a coin to the slot. A coin can participate in an insert event when it is close to a slot; then, one slot is required to participate, which will take the coin (if it is not filled with too many coins already, which we will discuss later).

The three kinds of interactions discussed above, characterize three different user scenarios[3]: the user inserting a coin, the user ordering the drink, or the user pressing the cancel button. Most of these scenarios are described

[3]In part 2, there will actually be a forth one, which has to do with the user inserting a cup to which the drink can be dispensed, which corresponds to the cup_in event. But for now, there is no way for the user to trigger this event. The resp. Output device will be added in part 2 of the model.

by the coordination diagrams, but as you could see from the forward references to the local behaviour, part of the coordination is coming from the local behaviour.

In order to get the complete picture, let us discuss the local behaviour of the different elements now. We start explaining the local behaviour of the Control, which is shown in Fig. 4.4: There are two transitions, both of which are always enabled. Both of them are bound to two event types in order to synchronize them as already discussed above. The top transition requires that the event pass and drink are always executed together – this way making sure that dispensing a drink and paying it always go together. In this event binding, no parameters are involved, and the action attached is an output statement only, indicating the class of the brewer that is involved – which is not interesting for now, but might be a bit more interesting once we introduce different kinds of brewers in part 3 of the model.

The bottom transition of the local behaviour of Control also synchronizes two different events: yyy and reset. yyy is one of the oddly named events, which were defined in the coordination diagram of Fig. 4.3. So, let us briefly explain it: the event type yyy is connected to the event type cancel with something that looks like an inheritance relation, but with a filled arrow head. This indicates a special kind of inheritance on events, which we call *extension*. Basically, this means that yyy is a cancel event, just with an additional parameter, *test* in this case. As far as coordinations are concerned, yyy is a cancel event – and cancel is called the *base* event type of event extension of yyy. The only difference is that partners that are interested in the additional parameter could use yyy in order to contribute or to access this additional parameters. We discuss this in much more detail later in Sect. 4.2.2.2. Altogether, the bottom transition synchronizes the cancel event (represented by its extension yyy in the net) with the event reset. It also contributes some message to the parameter of the extension yyy. The action does not do much: it prints out the value of the parameter of the yyy event.

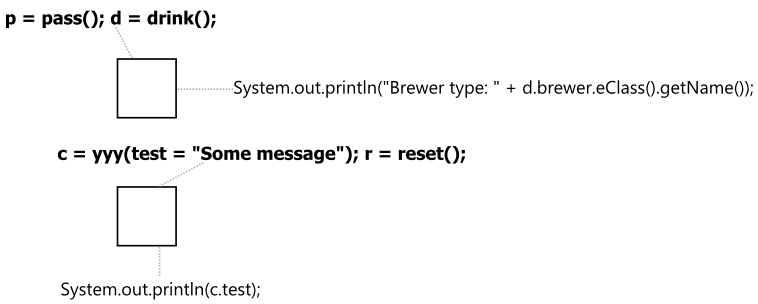

Figure 4.4: Behaviour of Control

Figure 4.5 shows the life cycle of the brewer. Also here, an oddly named is involved: bla, which is an extension of a drink event. Initially, a brewer is in state ready, where it can do a reset or a drink event – represented by the event extension bla. A drink event will bring the brewer to state brewing. From the state brewing, the brewer could do a cup_in even, which would bring it back to state ready. But, for now there is no element that would trigger a cup_in for the brewer. Therefore, for now all brewers will be stuck in state brewing after a drink event. The event binding for event bla has a parameter, which shows a more complex from of parameter assignment: `n.base.brewer = self()`. To understand this, we need to have a look at the event types in Fig. 4.3 again. Event bla refers with name `n` to another event extension blubs. This means that bla is not only an extension of event bla, but also an extension of the other event extension blubs, which in turn is an extension of drink. In general, an event extension extends exactly one event type, but it can extend any number of other event extensions. In order to refer to the parameters of other event extensions, the event extensions that are extended, are referred to by name – `n` in our case. We will see later that all the event extensions as well as the base event type from which an event extension is derived must be on a single line of the event type inheritance hierarchy. By using the name of an event derivation `n` in the parameter assignment, we refer to the event extension blubs. With `base`, we refer to its base event; and at last `brewer` refers to that parameter of the drink. Of course, we could have directly used the drink event here; we used these extensions only for discussing these concepts here. Also the action of the transition bound to bla prints some information on the classes behind the values attached to some of its parameters. Note that here the base and other extended event types and extensions are accessed by methods `n()` or base(); the reason for not using the parameter names directly here and use method calls instead are some lazy initialisation mechanisms for the event instances and their parameters, which we do not discuss here in more detail.

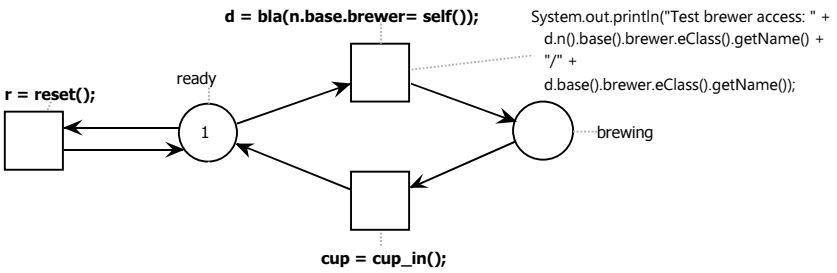

Figure 4.5: Behaviour of Brewer

At last, we discuss the behaviour of the Coin and Slot which are the only ones with real actions attached to them, which take care of inserting the coin and transferring the coin from the slot to the safe. The ECNO nets

for these two element types are shown in Fig. 4.6 and Fig. 4.7. The life cycle of a coin is shown in Fig. 4.6: A coin can be inserted and returned an arbitrary number of times, but once an inserted coin is passed to the safe, it remains there forever and cannot do anything anymore. In the event binding for the insert event, the coin assigns itself to the *coin* parameter. In the respective action, the coin removes itself from the slot it is close to (since it will be in the slot now – see local behaviour of the Slot). The coin also removes itself from the engine, which means that it will not be shown on the GUI anymore. Note that the engine can be accessed by using the predefined keyword `engine` in any behaviour. The action associated with the return_ is the opposite. The coin adds itself to the attribute slot again, since it is close to the slot again, when it is returned; and the coin will show in the GUI again, which is achieved by `engine.addElement(self())`. For the pass event, there is no action for the coin. But the coin assigns itself as coin parameter – so that the safe could receive this coin.

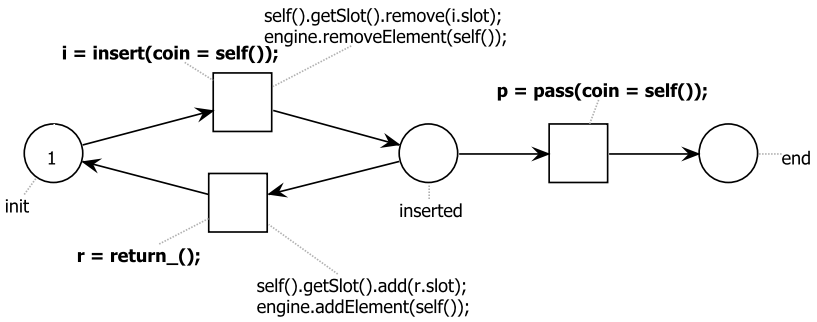

Figure 4.6: Behaviour of Coin

The life cycle of a slot is shown in Fig. 4.7. The top transition represents the insertion of a coin: In the event binding, the slot assigns itself to the slot parameter; note that the condition associated with this transition makes sure that a coin can be inserted only if there are less than two coins in the slot already; the action assigns the coin (which comes from the coin parameter of the insert event) to the slot's coin attribute. The middle transition represents passing a coin on to the safe; again, the slot assigns itself to the coin parameter, and in the action removes the coin from itself. The bottom transition represents returning all the coins contained in a slot, which is triggered by a reset event; the slot assigns itself to the slot parameter of the return_ event, and in the action, deletes all coins that currently are contained in the slot.

The attentive reader might have realized that we are still missing the local behaviour for two element types: the Panel and the Safe. We do not have models for them, because these elements have a default behaviour: they can participate in any event at any time and they do not have any

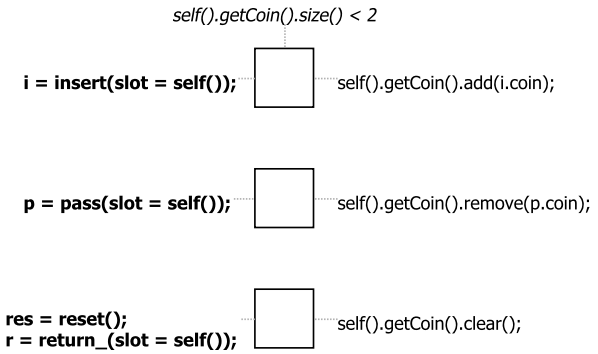

Figure 4.7: Behaviour of Slot

action. If not stated otherwise, this is the behaviour which is associated with any element type. Note that this way, the Safe actually does nothing at all; it actually does not even store the coins – they just vanish. And the behaviour of the panel will actually come from the user by clicking on the respective buttons. Since the local behaviour of the panel does not restrict the behaviour, all interactions that are possible for some event of type drink or cancel are shown as enabled buttons – if no such interaction is possible, the resp. button is disabled.

4.1.3 ECNO models: Part 2

In part 1 of our model, there is no output device yet. Therefore, eventually all brewers of the vending machine (part 1) will be stuck since they cannot dispense the coffee. They are waiting for a cup_in event forever after brewing the first drink. Part 2 of the vending machine remedies this problem in that it adds an Output element. The idea is that the user can put a cup in there, to which the coffee is dispensed, and he can remove the cup afterwards, which will be reflected by the the event type cup_in and cup_out.

Figure 4.8 shows the ECNO package for part 2 of the model, where the element type Output and the event type cup_out are new. The element type Output is a GUI element and the event types cup_in and cup_out are its GUI events, which means that the user can trigger these events here. The Output has a reference to a Brewer, which was defined in part 1 already. The small arrow icon in the top right of the element type Brewer indicates that this element type is *imported* from an other package. Note that also the event type cup_in is imported from a different package by such an icon; only the event type cup_out is new here. Note that we need to import the cup_in event to this package since the local behaviour of the element type Output of this package refers to it; if we did not import it, the local behaviour would not be able to resolve this event type.

There is not a lot of coordination going on in this part. The only required coordination in the global behaviour says that if an Output element is involved in a cup_in event, then there needs to be one Brewer that participates in it. This will be the brewer that dispenses the drink to the cup.

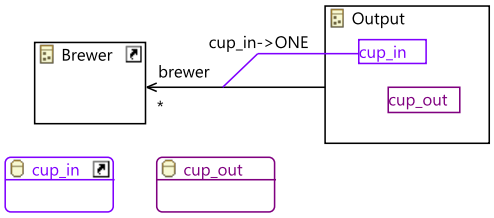

Figure 4.8: Vending machine, part2: Coordination

Figure 4.9 shows the local behaviour of the Output element: it consists of a cyclic succession of cup_in and cup_out events. But cup_out is an event that is local to the Output, which is not coordinated with other elements (see Fig. 4.8).

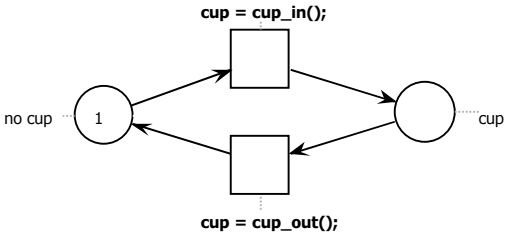

Figure 4.9: Behaviour of Output

4.1.4 ECNO models: Part 3

In part 3 of the model, we come to some more interesting concepts, where we actually specialize the brewers to two different kinds: we introduce a CoffeeBrewer and a TeaBrewer, which inherit from Brewer. Note that we had seen that in the Ecore diagram in Fig. 4.1 already; but in Fig. 4.10, this inheritance is also included to a coordination diagram, with the effect that the life cycle or local behaviour of a CoffeeBrewer consists of two parts: the local behaviour associated with the Brewer as discussed in Fig. 4.5 and the local behaviour associated with the CoffeeBrewer, which is shown in Fig. 4.11, and is discussed shortly. Similarly, the behaviour of a TeaBrewer consists of the behaviour of the original Brewer and the TeaBrewer.

Before discussing the local behaviour of the brewers and the notion of behaviour inheritance, let us discuss the additional event types that are shown in the coordination diagram of Fig. 4.10: There are two new event types,

coffee and tea, both of which are derived from the event drink from part 1, which is imported to this package. Note that the relation to event type drink is graphically represented by the usual arrow for inheritance in UML; and we say that event types coffee and tea *specialize* the event type drink. In contrast to event extensions, which we had seen earlier already, event types coffee and tea are new event types, both of which are compatible with drink. But coffee and tea are different and are not compatible with each other – an event cannot be a coffee event and a tea event at the same time. In our example, the coffee event and the tea event also have different parameters, but this is not what makes them incompatible (different extensions of the same event with different parameters are compatible). In Sect. 4.2.2, we discuss some more details of the rational behind the concepts of *specialization* and *extensions* of events. What is important for now is that the coffee and tea events should be different, and even if some specializations are added in later additions to the project, they are supposed to remain different (for this reason we do not allow multiple inheritance on event types).

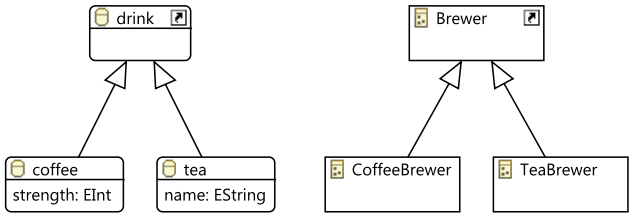

Figure 4.10: Vending machine, part3: Coordination

The life cycle of the CoffeeBrewer and the TeaBrewer are shown in Fig. 4.11 and 4.12, respectively. These behaviours are actually quite simple, since most of the local behaviour is inherited from the life cycle of the Brewer. The choices for all event types not mentioned in this behaviour are the ones of the Brewer. There is, however, one minor twist for both: The CoffeeBrewer has a transition with an event binding for coffee; since coffee inherits from drink, this is the choice taken when the CoffeeBrewer is asked to participate in a drink – of course this transition will be taken together with the transition bound to drink in the life cycle of the Brewer as shown in Fig. 4.5. This way, the CoffeeBrewer can actually not "do" an unspecific drink event, since the local behaviour of the CoffeeBrewer enforces it to "do" a coffee event; and it could definitely not be a tea event, since the tea event would not be compatible to the event that is required to happen – the coffee event.

The local behaviour for the TeaBrewer is basically the same, except that it requires the drink event to be the more specific tea event.

The interesting thing now is that, if an interaction is computed with a drink event from the panel of the vending machine, due to the coordinations, it will require a CoffeeBrewer or a TeaBrewer to participate: the CoffeeBrewer

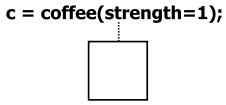

Figure 4.11: Behaviour of CoffeeBrewer

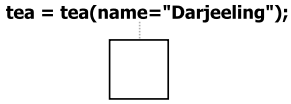

Figure 4.12: Behaviour of TeaBrewer

resp. TeaBrewer will turn this event into a coffee or tea event respectively. Resulting in different interactions, and once the "drink" button on the GUI gets enabled, it will actually show the type of the more specific event. This way, magically events turn out to be of the right type, and the panel does not even need to know which kind of event types are extending the drink event. Of course, it could happen that different parts of the interactions would require to turn the same event into different incompatible event types. In that case, the respective interaction is invalid.

Of course, the extended life cycle of an element type does not need to be that simple. It can also add additional restrictions on other events. The overall behaviour of an element would be all the life cycles of the element type hierarchy synchronized with each other and events appropriately specialized. An element type that inherits from another element type could also have event types that did not occur in the super types at all, in that case, the super type would not be involved in that part of the behaviour.

Note that this form of inheritance on the one hand side restricts the local behaviour for the element for the event types that already existed in the super type. But, since there can be also new event types that did not exists in the element's super type, there can be also some additional behaviour. Moreover, ECNO nets have some additional concepts, which allow us to override or even completely ignore behaviour of the super types. We discuss this in a the variation of part 3 below.

4.1.5 ECNO models: Part 3 (variation)

In order to discuss some more concepts and effects that have to do with inheritance, we discuss a variation of the model of part 3 that we have discussed in Sec. 4.1.4. Note that the variations are mostly of technical nature just to make our point and explain some additional features of inheritance in ENCO.

Figure 4.13 shows the coordination diagram of the variation of part 3. In

addition to the earlier version of Fig. 4.10, the cup_in, reset, and cancel event are imported, so that they can be used in the ECNO nets for CoffeeBrewer and TeaBrewer, and there is another new event kick which inherits from the cancel event. This kick event represent kicking the vending machine, for making the tea brewer actually brew the tea; in the model for the local behaviour of the tea brewer, we build in a flaw that will require the user to kick[4] the vending machine in order to proceed.

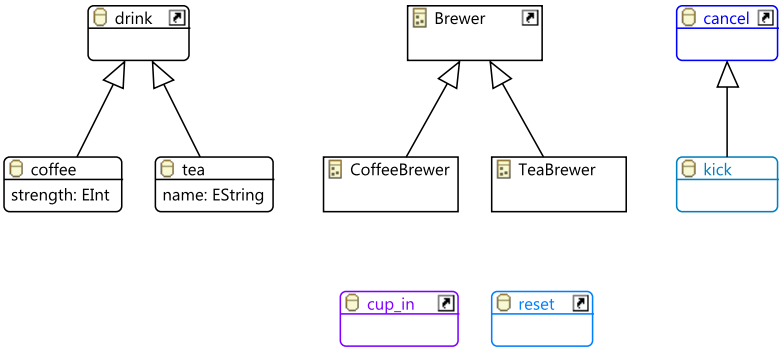

Figure 4.13: Vending machine, part3 (variation): Coordination

The more interesting changes and additional features are in the local behaviours of the CoffeeBrewer and the TeaBrewer. We start with the CoffeeBrewer since it is slightly simpler. The local behaviour of this variant of the CoffeeBrewer is shown in Fig. 4.14. The coffee now has an action, which is `parent.drop();` this action results in dropping the action that would be executed for the choice of the super element types that is involved with this event. Note that dropping the action of the super element type means, that neither the Petri net transition fires nor its Java action is executed, but the parameters contributed by the choice of the super element type – if there were any – remain available.

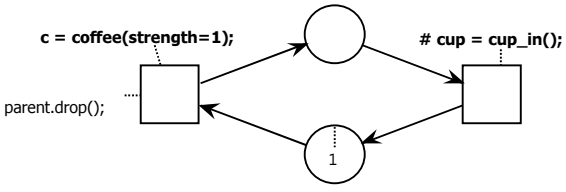

Figure 4.14: Variant behaviour of CoffeeBrewer

Note that due to dropping the parent action, the net of the parent Brewer (see Fig. 4.5) would still expect a coffee event, whereas the net of the CoffeeBrewer would be ready for the cup_in event already. And this would block

[4] A phenomenon, which I learned about from American movies.

the CoffeeBrewer forever since the different parts of its local behaviour in the type hierarchy are in an incompatible state. First and foremost, this should show that dropping the behaviour of the parent should be done with great care. Second, we show how to work around this problem with another feature, which completely ignores or overrides the behaviour of all its super element types. This is indicated in the event binding of the choice reps. the transition representing it. A hash sign "#" in front of the event binding indicates that, when the element takes this choice, all the behaviours and possible choices of the super types are completely ignored. In the local behaviour of the CoffeeBrewer shown in Fig. 4.14, we use this in the event binding for cup_in; this way, we can resolve the problem of the net of the super element type being in a state where cup_in would not be possible. And, after executing it, both nets would be in a compatible state again, both could execute a coffee event again.

At this point, we would like to point out another important feature related to inherited behaviour: The Brewer can also be involved in a reset event, which is possible only when the net is in the initial state. Now, the local behaviour for the CoffeeBrewer does not refer to the reset in any binding. Therefore, the CoffeeBrewer does not contribute to the reset in any way – but it does not restrict it either. It would be the same as for the Brewer. Due to the `parent.drop()` action, which we had discussed above, the net for the Brewer would never leave the initial state. Therefore, the CoffeeBrewer will always be able to participate in a reset event – in contrast to a Brewer and the TeaBrewer, which we discuss later.

Note that, by default, the actions of the different choices in the type hierarchy of an element are executed bottom up: The action of the choice of the most specific element type is executed first, and then sequentially the actions up the hierarchy are executed, unless one of the actions explicitly requires dropping the parent's behaviour with a `parent.drop()` instruction in the action. In some actions, we might want to execute the actions in a different order. To this end, an action can executed `parent.execute()` at any point; this initiates the execution of all actions of the parent (and its ancestors) at this point; and after that command, the action can be sure that all ancestors actions in the type hierarchy have been executed or dropped – if some of the ancestor decides to drop its parent's behaviour. Anyway after a `parent.execute()` an action would be sure that no actions of the ancestors would be executed later anymore. Technically, ECNO nets do not prevent an action to call both, `parent.drop()` and `parent.executed()`; the action could even issue these commands several times. It is the first call, that actually determines what would be done – all later calls in an action do not have any effect. By using `parent.drop()` and `parent.executed()` the actions have control over which of the parents action should be executed and when. But, the control of this execution order is with the sub-types – a super type does not have any influence on whether and when it is executed

by the behaviour of its sub-types.

Next, let us discuss a variation of the local behaviour of the TeaBrewer, which is shown in Fig. 4.15. Basically, the tea brewer requires an additional event kick to happen between a tea event and a cup_in event – representing the flawed tea brewer that needs a kick to continue brewing. The brewer, however, is not a GUI element type and thus would not be visible to the user – the user could not kick it. That is why we choose to specialize the kick event from the cancel event. This way, it would be visible at the panel as a specialized cancel event – and at the same time we require the TeaBrewer to participate in a reset event. Since we want the reset event to participate in this choice, this choice needs to ignore (indicated by the leading hash sign "#" in this event binding) the possible behaviour of the super type – the reset event would not be enabled in the Brewer at that point. But this does not do any harm, since this reset event at this point is completely independent of the Brewer. Once the TeaBrewer was involved in a kick event (along with the reset event), it would be ready for cup_in.

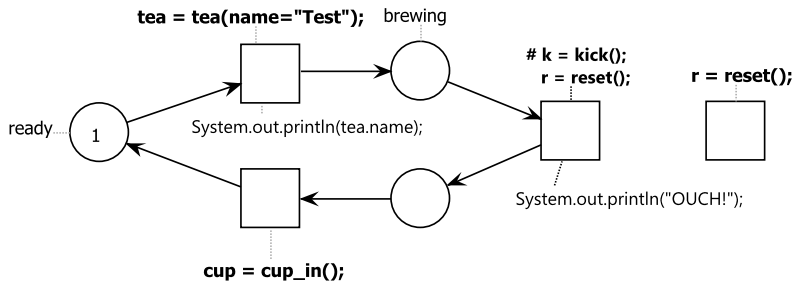

Figure 4.15: Variant behaviour of TeaBrewer

There is one last subtle issue in the behaviour of the TeaBrewer: the transition bound to the reset event. Such a transition was not needed in the local behaviour of the CoffeeBrewer. In contrast to the CoffeeBrewer, however, the TeaBrewer refers to the reset event in one of its binding. Therefore, it would be only enabled when these transitions are enabled. But, the reset event should also be possible when the Brewer is in its state ready. The transition bound to reset only in the TeaBrewer make this possible.

This shows that you need to be very careful about any events of the super types. As soon as you refer to an event type in the local behaviour at all, you have full responsibility not to restrict it further than necessary. In our case, this is achieved by a completely detached transition bound to this event – which leaves the full control of such a single event to the behaviour of the element's super type.

4.2 Concepts of inheritance

The example in Sect. 4.1 covered the most important concepts of inheritance already, and we have loosely explained the most important concepts of inheritance.

In this section, we discuss the concepts of inheritance more systematically and in more detail. In Sect. 4.2.1, we discuss inheritance on elements and how it affects their behaviour. Therefore, we call this behavior inheritance. In Sect. 4.2.2, we discuss inheritance on event types – more specifically, we discuss specialisation and extension of event types.

In many ways, ECNO and its inheritance mechanisms on event types resemble concepts of aspect-orientation [28, 40] and subject-orientation [19]. The relation to aspect-orientation and how the way of aspect- and subject-oriented thinking has influenced some design choices of the ECNO is discussed in Sect. 4.2.3.

4.2.1 Behaviour inheritance

In this section, we discuss the effect of inheritance on element types on the behaviour of elements. This mostly affects the local behaviour of an element, but could also affect the coordination among different elements.

In ECNO, *inheritance* on element types is represented by the fact that each element type can have one other element type as a *super type*. ECNO supports only *single inheritance* on element types, which means that one element type can have at most one (direct) super element type. While multiple inheritance on element types could make sense and could probably given a reasonable semantics in ECNO, it complicates the definition of inheritance a lot (see Sect. 4.2.1.4 for some more details). Therefore, the current version of ECNO does not support multiple inheritance on element types. ECNO, however, works together with underlying object-oriented technologies supporting multiple inheritance. In addition to each element type having at most one direct super type, there are two other requirements: the inheritance relation on element types must be acyclic[5], and that the underlying class of the super element type must be a direct or indirect super class of the class underlying the element type itself. Note that ECNO allows jumping over some super classes of the object-oriented model. The classes that are jumped over, would not contribute to ECNO's behaviour.

Graphically, the inheritance relation is represented in ECNO coordination diagrams in the same way as inheritance in UML class diagrams (see Fig. 4.10). The more interesting question is how the inheritance on element types affects the behaviour of the respective element type, which concerns the coordinations as well as the local behaviour.

[5] Actually, this requirement carries over from the object-oriented technology.

4.2.1.1 Local behaviour

We start explaining the local behaviour of an element of type t_n, which inherits directly from element type t_{n-1} and indirectly from element types $t_{n-2}, \ldots, t_1$. The element would have a local behaviour for each type $t_1, \ldots, t_n$. The overall local behaviour would, basically, be as follows: When the element participates in an event of type e, basically, each type t_i would be required to make a choice c_i which is compatible with the local behaviour for type t_i. We call these the c_i the stack of choices of the element.

But, as we have seen before, a choices c_i for an element type t_i might require the participation of some additional events. For each of these additional events e_k, each choice c_i of the stack would need to contain this event e_k, too – except in the following three cases:

1. The local behaviour for type t_i does not refer to the event type of e_k at all.

2. None of the element types $t_1, \ldots t_i$ do have a coordination set for the event type of e_k.

3. If the choice on level t_i explicitly requires to ignore (cut off) the local behaviour of the super types, the choice for all levels above are empty choices.

The first condition allows an element type t_i not being involved in some types of events. In our example, of Sect. 4.1.4, we have seen that the local behaviour of the coffee and tea brewer were not involved in the reset and cup_in event at all. Therefore, this was left to the local behaviour of the Brewer; the local behaviour of the CoffeeBrewer or the TeaBrewer were not involved in the reset events at all.

The second condition says that, if an event type is defined for the first time for the element type t_{i+1} in the element type hierarchy by defining a coordination set for that event type for that element type, all the element types above do not need to participate in that event type.

The third condition deals with the case that a sub-type explicitly cuts off the local behaviour of its super types, which is discussed in more detail in Sect. 4.2.1.2.

Up to this point, we have discussed local behaviour that is not parallel. As we had mentioned earlier (see Sect. 2.2.4), it is possible that the local behaviour of an element type can participate in more than one choice at the same time. We called that *parallel behaviour*. This is also possible when element types inherit from each other. In that case, ECNO requires that for each parallel choice there is a stack of choices, meeting the rules above, and on each level all the choices are possible to be executed in parallel according to the model. In a way, inheritance on local behaviour and parallel behaviour are orthogonal to each other.

4.2.1.2 Explicit control of participation of super types

In the variant of Part 3 of our example in Sect. 4.1.5, we have seen that ECNO provides some explicit mechanisms for controlling whether, how, and when the super type of en element type should be involved in the local behaviour. The choice on each level controls in which way to involve or not to involve the higher levels.

Basically, there are two different mechanisms for a choice to control the participation of the local behaviour of the higher levels in the ancestor hierarchy:

1. The choice can require to completely ignore the local behaviour of all the super types of the element type. In ECNO nets, this is indicated by the hash sign (see Fig. 4.14 for an example) in front of the respective event binding.

2. The choice can allow to synchronize with the local behaviour of the super types. In that case, the choice will only be possible if also the super types can participate in it. But in its action, the choice can execute a `parent.drop()` command, to the effect that the actions of the choices of the super types are never executed. This way, the local behaviour of the super type does not contribute any changes; but it contributes parameters and its synchronisations.

When a choice of an element type with inheritance is executed, this means for each choice in the stack of choices each action is executed, starting from the lowest element in the inheritance hierarchy up to the highest. This order can, however, be changed by the actions that are lower in the type hierarchy: By calling `parent.execute()`, they can make sure that, at this point, all the behaviour of the stack above it are executed – the detailed order controlled by the resp. actions on the higher levels. Note that if `parent.execute()` and `parent.drop()` occur in an action, or if they occur several times, only the first time invoking these commands will have an effect; all further calls are ignored.

In general, ignoring or dropping the local behaviour of the super types should be used with great care, since it easily results in the life cycles on the different levels being "out of synch", which might result in unexpected behaviour or, more likely, in no possible behaviour at all after that point. In our example of Sect. 4.1.5, we actually needed to take some extra measures to work around blocking the overall local behaviour of the CoffeeBrewer and the TeaBrewer.

4.2.1.3 Extending coordinations

Up to now, we have discussed inheritance on element types and the resulting local behaviour for elements of the respective type. Inheritance on element

types can have an effect on the coordination with other elements too. The derived element type can have additional coordination sets for some events. Here, we briefly discuss the effect of these additional coordination sets.

Basically, there are two possibilities: The coordination set could refer to an event type for which no coordination set existed in the super element types. In this case, the super types are not concerned in this event type at all; and the new coordination set or the new coordination sets define the coordination for this new element.

The second possibility is that the element type has a coordination set for an event type for which the super element types had a coordination set already. In this case, this new coordination set, adds one additional choice of coordination for this element.Therefore, this extends the possible interactions for this element, since the coordination for an element and a given event has the choice between all the coordination sets that are associated with the event's type. Effectively, the element type has all the coordination set of all its super types.

Note that the basic mechanisms of inheritance restricts the local behaviour of the element. This does not apply to coordination when adding coordination sets in the type hierarchy. Therefore, the way we deal with additional coordination sets is debatable. We will discuss some alternatives and possible extensions in Sect. 4.2.1.4 below. For the current version of ECNO, we stick with this simple semantics, since in most of our examples, derived element types do not have additional coordination sets at all; when they exists, they refer to new event types. Therefore, we deemed that this was a less important issue.

4.2.1.4 Discussion and possible extensions

In the previous sections, we have discussed inheritance on element types and what inheritance means for the behaviour of the elements. As discussed above, some things could have been defined a bit more general or given a slightly different semantics. In this section, we briefly discuss some of the alternative choices or generalisations.

Up to now, ECNO does not allow multiple inheritance. Technically, it would not have been difficult to implement multiple inheritance: instead of a stack of choices for the different behaviours of the element types an element inherits from, we would need to maintain a network of such choices. The reason, ECNO does not support multiple inheritance is a conceptual one: As discussed earlier (see Sect. 2.2.7), the execution of an interaction should be deterministic (under the mild assumptions that each local behavior makes local changes only). As soon as we introduce multiple inheritance and allow commands dropping or executing the behaviour of the parents as discussed in Sect. 4.2.1.2, interactions would not be deterministic any more: to see this, let us have a look at a simple example, with four element type A, B, C, and D,

where B and C inherit from A and C, and where D inherits from both B and C. Now let us assume that each of these element types have a local behaviour for some event type e, each of which has some visible effect. Furthermore, assume that the action of B drops the behaviour of its parent, whereas C would require to execute the behavior of its parent in the first place. In that case, B and C could not both have their way. Second, it would depend on the order in which the local choices of B and C are executed whether the choice for A is executed or not. Of course determinism, could still be achieved by introducing some order in which the actions of B and C are supposed to be executed. The question, however, would be where this order would come from, and defining such an order explicitly might be very confusing for a modeller. And this would still leave the problem that B and C cannot not both get their way (dropping the parent's choice or not). Of course, this concrete problem could also be resolved by removing the feature to drop the parent's choices from ECNO. But, this would still leave the problem that the choices of B and C are executed in an arbitrary order. The easiest and clearest way to make sure that the execution of interactions are deterministic was excluding multiple inheritance on element types from ECNO. Only very compelling reasons – i. e. examples that could not adequately be modelled without multiple inheritance – could result in extending ECNO supporting multiple inheritance on element types.

We had also seen that, if a derived element type introduces a new coordination set for an event type that exists in a super class, this introduces a completely new choice of coordinations. This is in contrast to what we expect from a derived element type: it should specialize behaviour and not extend it (at least in the standard cases). In all of our examples, however there was no need to add this kind of new coordination set. We needed to add coordination sets only for event types that were added for this element type at this level. We could have restricted ECNO in such away, that element types can add coordination sets for new event types only; but this appeared to be too restrictive. One possibility of avoiding the above problem would be that a coordination set of an element is derived from a coordination set of its super types. This way, an existing coordination set is extended by an additional choice. For a definite decision, we would need more and more realistic examples.

Another decision we had taken for the local behaviour of an element type was that event types that do not occur in any event binding are ignored, and the behaviour on that level does not need to participate in that event. An other design choice would have been that such an event is always blocked. For either of these two semantic choices, the models of the local behaviour could always be changed in order to obtain the desired behaviour with the choice of the respective semantics. Therefore, this design choice is about convenient and intuitive models only. We had the feeling that, in our examples, ignoring event types that are not mentioned at all gives in

the local behaviour result in more "natural" models (ECNO nets) – but a definite answer to which of these choices is more appropriate require more research and larger case studies.

4.2.2 Event inheritance

In this section, we discuss *inheritance* for event types. In particular, we motivate why we need to have two different kinds of inheritance for event types, which we call *specialization* and *extension* of event types.

4.2.2.1 Event specialization

We start with discussing the notion of *specialization of an event type*. In our example, we have seen in Fig. 4.10 that the event types tea and coffee specialize the event type drink. This way, the event of preparing some drink is specialized to the event of brewing a coffee or to brewing a tea.

Through a specialization, an event type can be equipped with an additional parameter, which, in our example, was the strength for the coffee event type and the name of the sort for the tea event type. More importantly, however, two different specializations of an event type are *not compatible*, which means an event cannot be of these two types at the same time. An event of type coffee cannot be of type tea at the same time – surely at least tea enthusiasts would agree to that. But of course, an event of type coffee is of type drink at the same time. When expecting an event of type drink, the actual event might be of a more specific type coffee or tea – but not both at the same time.

The idea is that specialization makes things more specific, and different specializations make things different and incompatible. For that reason, ECNO does not allow multiple inheritance with respect to specialization. With respect to specialization, each event type has at most one *super event type*. Of course, there can be indirect ancestors in the specialization hierarchy. Two event types are *compatible* if one of the event types is an ancestor of the other in the specialization hierarchy.

If we allowed multiple inheritance – or multiple specialization – on event types, it would be possible to make coffee and tea compatible by adding a new event type, say confusion specializing both of them. This way, later additions to a model of our vending machine could make things compatible, which were assumed to be incompatible in the basic model. In the least, this would introduce a lot of confusion. Therefore, ECNO does not support multiple specialization on event types.

Of course, there are situations when things should not become incompatible when extending them in different ways. This is why there is a second notion of inheritance on event types, which is discussed later in Sect. 4.2.2.2.

First, let us discuss the effect of specialization on the behaviour, i. e.

its effect on the coordination and the local behaviour. The idea is simple: whenever a condition of a coordination annotation or a synchronisation in the local behaviour is checked for an event of some type $t1$, it is checked whether the event type $t2$ of the condition is compatible with event type $t1$ and whether $t1$ is at least as specific as $t2$. This way, the most specific event type in the coordination conditions defines the actual event type of the event. Note that this might rule out some interactions: for example, if one partner's condition would imply the event to be a coffee event and another condition would require the same event to be a tea event, this would not be a legal interaction.

The current version of ECNO has one important restriction concerning event bindings or the types of the events involved in a choice, though: The event types of each event binding or choice must have different *top-level event types*. For an event type, its associated top-level event type is the top-most ancestor in the event type hierarchy. In our vending machine example, this excludes an event binding synchronizing a coffee and a tea event in the local behaviour for an element type.

Without this restriction, the semantics of ECNO together with inheritance would be very complicated, and there would be many design choices, the resolution of which would look a bit arbitrary – at the current stage. Moreover, implementing the ECNO engine in a half-way efficient way would have been very tricky. Therefore, ECNO makes this assumption. Since there are relatively simple ways (a kind of design patterns for using ECNO) to indirectly synchronize event types that have the same top-level event types, this restriction does not reduce the expressive power of ECNO – removing the assumption is syntactic sugar and, therefore, not worth the extra effort at the current stage.

Since all event types involved in an event binding or choice must have different top-level event types, ECNO does not have a built-in unique top-level element, which would correspond to the top-level class `Object` in Java. Each event type that does not explicitly refer to a super event type is a top-level event type and all top-level event types are incompatible to each other.

4.2.2.2 Event extensions

In the previous section, we have seen that two different specializations of the same event type will always be incompatible. In some cases, however, different parts of an interaction might want to extend an event in different ways, just to contribute parameters that are necessary for their purpose. But, they do not want to make the actual event type different. To this end, ECNO supports another notion of inheritance, which we call *event type extension* or, for short, *event extension*.

An *event extension* does not define a new event type – it just extends an existing one. From the point of view of coordination and synchronisation,

an *event extension* behaves as the event type it extends; we call this event type the event extension's *base event type*. The only difference is that the extended event type carries some additional parameters, which are ignored by partners using the base event type or other extensions of it.

In Fig. 4.3 on page 69, we have seen examples of event extensions already: yyy, bla, and blubs. The strange names should help emphasizing that these extensions are not event types in their own right. The event type extension yyy still represents a cancel event, and bla and blubs still represent a drink event. In a way, an extension is like an addition to the original event type; therefore, the base event type of an event extension is graphically represented by a inheritance relation with a filled arrow head (see Fig. 4.3), where the filled arrow head should resemble the filled diamond of a composition.

Note that an event extension must refer to exactly one base type. But an extension can further extend any number of other event extensions. These are graphically represented by arrows, which resemble a reference in class diagrams; since an event extension can extend more than one other event extension, these references carry a name, which can be used for referring to the parameters defined in the respective event extension. In order to uniquely identify the respective extensions (see more details in Sect. 4.2.2.3), the names of these references of an event extension to the extended extensions need to be different. Moreover, the base event types of the extended extensions must be the base event type of the extension itself or one of its ancestors.

As said above, concerning coordination and synchronisation, event type extensions do behave as their base event type. The added value is that they allow to contribute new parameters and to use these parameters in the expressions of parameter assignments, conditions and actions. This will be discussed in more detail in Sect. 4.2.2.3 below.

The idea of event type extensions, where one extension is not in conflict with other extensions, is very similar to the idea of subject- or aspect-orientation: Basically, an event type extension can be considered as one aspect of the event type not conflicting with other aspects for the same event. We will come back to that in Sect. 4.2.3.

4.2.2.3 Assigning and using parameters in ECNO nets

In the previous subsection, we have discussed the different forms of inheritance for events. The two mechanisms of specialization and extension, in particular, allow the definition of parameters for event types and event type extensions. These parameters can be contributed and used in the local behaviour of the respective elements, which is defined by the possible choices. As stated earlier, there are different ways of defining the local behaviour for element types. But, the main notation used for modelling local behaviour in this technical report are ECNO nets. In this section, we discuss how a

choice can contribute parameters to an event, and how these event parameters can be used in expressions, conditions and actions of a choice – using the concrete syntax of ECNO nets.

The basic mechanisms have been discussed already in Sect. 2.2.4 in the absence of inheritance. Therefore, we focus on the additional mechanisms concerning inheritance on event types. For completeness sake, however, we start with the basic mechanisms anyway.

We start with the mechanisms for assigning an actual parameter to an event, which is done in the event binding. Let us have a look at the the example from Fig. 4.5 on page 72 again. The event binding in this example read: `d = bla(n.base.brewer = self());` This binding refers to the event extension bla, with the base event type drink. In parentheses behind the event bla, there is one parameter assignment, where the left-hand side refers to a formal parameter of the event and the right-hand side refers to some expression; the expression, would evaluate to some value, which during the event binding is assigned to the respective parameter. In general, there could be any number of such parameter assignments for each event in such an event binding, which are separated by commas.

First let us discuss the left-hand side of the assignment, which refers to a formal parameter of the event. In our earlier examples from Sect. 2.2.4, this was just a name of a parameter of the respective event type. Now, we use the dot notation along with some additional keywords to denote specific formal parameters. In our above example `n.base.brewer` refers to the event extension with name `n`, which bla extends (see Fig. 4.3 on page 69). In this example, this refers to extension blubs; `base` is actually a keyword, which refers to the base event type of extension blubs, which is event type drink. At last `brewer` refers to the formal parameter `brewer` of the event drink. So, this is the parameter to which the value on the right-hand side of the parameter assignment will be assigned. Basically, ECNO uses the dot notation of object-orientation to navigate to the respective formal parameter. The last sections in such a qualified name is always a name of a parameter of the respective event type; the sections before navigate to respective event type or event type extension staring from the type of the event – using some names of the respective extensions or one of the two keyword `base` and **`super`**: For an event type extension, `base` refers to the respective base event type; for an event type, **`super`** refers to its super event type.

Note that explicitly navigating to the super event type by **`super`** is necessary only in the case of parameter shadowing (masking): i. e. when a specializing event type defines a parameter with the same name as one of its super types. In that case, `name` refers to the parameter of the event type itself and **`super.`**`name` to the parameter of the super type of the event with that name.

The actual parameters of all the events of an event binding can be ac-

cessed in a similar way. The event itself is accessed by the name it is assigned to, which, in the above example, would be `d`. From there, the name of the parameter can be used to access the value – if there was no parameter assigned, the value would be **`null`**. The navigation to the base type, super type, and the extensions, however is slightly different: This navigation is done by method calls `base()`, `super_()` and `name()` on the respective event, respectively, if `name` is the name of the respective extension. In the example of the brewer from Fig. 4.5, the expressions `d.n().base().brewer` and `d.base().brewer` are used to access the value of the parameter `brewer` of the drink event. Note that, in this example, both references actually refer to the same parameter.

Note also that the method for accessing the super type of an event is `super_()` and not **`super`**`()` since **`super`** is a keyword of Java. As for the navigation in the assignment, `super_()` would be needed only when the super and sub-type define parameters with the same name (shadowing/-masking).

The navigation in expressions via method calls instead of merely using their name has efficiency reasons; this way, parameters of extensions that are not accessed do not need to be prepared for access. The respective method calls implement a lazy access mechanism – making the actual value accessible the first time when this method is called. This way, the code generator for ECNO nets, basically, does not need to parse the code snippets of the ECNO nets at all. The code snippets of ECNO nets are, basically, copied to the generated code.

Note that, in earlier versions of ECNO, parameters were not assigned by referring to their name, but by providing them in a specific order. In the context of specialization and extensions, however, there is no canonical order on the parameters of an event anymore. Therefore, the explicit reference to formal parameters was introduced; this has the nice side effect that an event binding of ECNO nets needs to mention only those parameters of an event that it contributes to, which typically is only one ore two. The current version of ECNO, however, provides a compatibility mode which still uses the variable order. This can be used if the local behaviour does not refer to event type extensions. But, we do not discuss the details here and discourage using this outdated syntax, since it will eventually be removed from ECNO.

4.2.3 ECNO and aspect- and subject-orientation

In this chapter, we have introduced notions of inheritance for element types and event types. And, at least on a first glance, the notions of inheritance on element types and event types seem to be different. For event types there are two different notions of inheritance, *specialization* and *extension*, whereas for element types, there is only one notion of *inheritance*.

In this section, we discuss this mismatch a in more detail. We start with

the discussion of inheritance on event types again. Basically, a specialisation of an event type makes the event type more specific and two specializations of an event are different or incompatible. This is why multiple inheritance does not make much sense, since multiple inheritance would make things that supposedly were different equal again. By contrast, event type extensions add additional information or structure to the type, without making it incompatible with other extensions of the original event type or even specializations of the event type. Therefore, event extensions are very much in the spirit of subject-orientation, where different "subjects" could have different perceptions of an object [19]. In terms of aspect-orientation, the extensions could be considered to be an aspect of the original concept.

Now, the question is: Don't we need these two analogous concepts of inheritance also for element types. And the answer is: yes we do need them. And actually, we can easily express both of these notions: inheritance on element types is actually specialisation. So where is extension then? An extension can be realized by a composition relation (remember that we used a filled inheritance arrow for graphically representing event type extensions, in order to stress its similarity to compositions), which adds the additional information to the original object. Of course, this requires a slight overhead for properly maintaining this structure and for accessing the additional information by explicitly navigating to the compositions. But, this is syntactic sugar only. Actually, in the early precursor of ECNO that we used in an ad-hoc way to distill the essence of business process modelling (AMFIBIA) [2], we had an explicit aspect-of relation that explicitly reflected extensions of element types. When distilling the essence for ECNO, it turned out that this was, basically, a composition – used in a specific pattern. Since ECNO is about distilling the essence of coordination, we decided not to make this kind of relation an explicit concept anymore.

This, however, leaves the question: Why do we need the two different concepts of inheritance on event types then? The answer is quite simple: on event types, we do not have compositions and there are no explicit means for manipulating and building up complex events. And since events are supposed to be volatile and not supposed to be explicitly manipulated, it would not be in the spirit of events to add such mechanisms. Such mechanisms would change the nature of events and turn them into objects. The only way to build up events are parameter assignments in event binding which exploits the event type system to make their values available to the right partners. In a way, event type extensions can be considered to be a very controlled way of building up compositions of events and for accessing parameters of these components in a convenient way.

Chapter 5

Using the ECNO Tool

In the previous chapters, we have discussed the motivation, concepts, and the notation of ECNO for modelling the behaviour of systems on top of structural object-oriented models. Though we have used examples that are deployed together with the released ECNO Tool for explaining the features, the focus was not on the use of ECNO Tools, its editors for the different kinds of models and configuration files or the code generator. And we did not discuss the programming framework for extending the ECNO GUI or for implementing a new one.

In this chapter, we discuss more details on how to use the ECNO Tool and how to used the programming framework of ECNO. Moreover, we give a brief overview on the architecture and design of the ECNO Engine, so as to better understand how to properly use the ECNO Engine. A detailed discussion of the internal implementation of the ECNO Tool, however, is beyond the scope of this technical report.

Section 1.3 gave quick overview of how to use the ECNO Tool already. But, in order to be self-contained, we will repeat all the necessary steps in this chapter – and adding more details, which are for example needed to build up applications from different Ecore and ECNO packages, which, in particular, requires to import types from one package to another.

Concerning the modelling tools and code generation, we restrict our discussion to ECNO coordination diagrams on top of Ecore models and to ECNO Nets for the local behaviour. But, we will give a brief introduction on how local behaviour could be manually programmed in Sect. 5.5.4. For more information on how to obtain and install the ECNO Tool, we refer to Appendix B.

5.1 Creating models and instances

We start with discussing how to create the actual models with EMF and the ECNO Tool.

5.1.1 Ecore models

As mentioned above, the ECNO models that we discuss here are based on Ecore models. Therefore, we briefly explain how to create and edit Ecore models first. Concerning the Ecore models, there is nothing specific to ECNO at all. Therefore, we refer to the EMF book [8] and the online documentation of Ecore Tools (`http://wiki.eclipse.org/Ecore_Tools`) for more detailed information.

Concerning the configuration of the code generation from Ecore models via the so-called gen model, ECNO makes some assumptions, which are discussed in Sect. 5.2.1.

There are no specific requirements from ECNO's side which editor you should use for creating your Ecore models. If you are new to EMF, we suggest to use the "Diagram Editor for Ecore", which will part of your Eclipse if you follow the installation instructions of Appendix B.1.

Typically, you would first create a new EMF project with the "Empty EMF Project" wizard. This creates a new project in your Eclipse workspace, which is properly set up for using EMF models and for generating code from these models. For some minor usability issues[1] concerning the EMF code generator, we strongly[2] recommend that you start with creating the Ecore models, and generate the code from them before generating code with ECNO Tools.

In the EMF project created by the "Empty EMF Project" wizard, you will find a folder called "model", which is where the new Ecore model should go. You can create such a model with the "Ecore Diagram" wizard (in category "Ecore Tools"), which will create two files. The file name with the extension ".ecore" is the actual Ecore model, whereas the file with extension ".ecorediag" is the diagram information, which stores the layout information such as size, positions, and colours of the different model elements in the diagram. Once you create such a diagram, the wizard will automatically open it with the "Ecore Diagram Editing" tool. The use of this editor is mostly straight-forward, and we do not explain it here. For more information, see the online documentation for Ecore Tools at `http://wiki.eclipse.org/Ecore_Tools`.

One important feature that you will need for building ECNO applications from different packages, is the feature that allows you to *import* classes and data types from other Ecore models. This can be done as follows: If you have opened the diagram you would like to import to, you would choose "Create Shortcut..." by right-clicking in the canvas of the diagram and selecting "Create Shortcut..."; in the opened "Select model element" dialog,

[1] If a file "plugin.xml" exists in a project already, the package extension of the generated code will not be added to the "plugin.xml". In these cases, the extension would need to be added manually to the "plugin.xml", which is a bit tricky for people new to EMF.

[2] Unless you have much experience with using EMF, always follow this recommendation.

you are now presented with a browser that allows you browsing the projects and models in your workspace. Use this browser for navigating to the class that you would like to import and select it. Then, this class will be shown in your diagram with the usual graphical representation for classes, but with an additional "import icon" in the top-right corner, which should remind you that this is an imported class.

You can then add references to this imported class, and you can also inherit from this imported class. In principle, it would also be possible to add some features (attributes, references, or inheritance relations) to the imported class. Since this actually changes the model from which this class is imported, you should NOT do this, unless you know exactly what you are doing.

5.1.2 ECNO coordination diagrams

Next, we explain how to create and edit coordination diagrams on top of Ecore models with the editor of the ECNO Tool. These ECNO models could be created anywhere – they could even be created in a project different from the one where the Ecore model resides in. But, for small projects, the ECNO models could go to the "model" folder of the EMF project, as created in Sect. 5.1.1.

Note that we do not bother to explain the standard features like opening a diagram with the editor, using the undo/redo features, or saving diagrams. The ECNO coordination diagram editor, follows the standard principles for Eclipse editors – mostly, because, it is a graphical editor generated using GMF [14]. The focus of the following discussion is on issues that are specific to ECNO and some of the advanced features of the editor.

5.1.2.1 Creating and setting up ECNO coordination diagrams

In order to create a new ECNO Coordination diagram, you can use the "ECNO Coordination Diagram" wizard. Similar to Ecore Tools, this will create two files: the file with extension ".ecno" is the actual ECNO model, the file with extension "ecno_diagram" contains the diagram information. The use of the "ECNO Coordination Diagram Editor" is similar to the use of the editor for Ecore diagrams. There are some specialities, however, which we discuss below.

First of all, each ECNO coordination diagram has an underlying Ecore model, which we recommend to set first. In an ECNO coordination diagram which is opened in the graphical editor, right-click in the canvas and select "Load Resource..."; in the "Load Resource" dialog that then opens, select "Browse Workspace" and in the opened "File Selection" dialog, select the Ecore model file. Note that this does not make this model the underlying Ecore model yet; this makes the Ecore model known to the editor. Now,

you can go to the editor's properties view and click on the drop-down menu for "EPackage". Here you should see the name of the Ecore model you have loaded before (maybe, you can see some others too). By selecting an Ecore package here, you make this package the underlying Ecore package of the ECNO model. The "ECNO Coordination Diagram editor" will – if it was empty before – create element types for each of the underlying classes of the Ecore model, along with the references and inheritance relations between them. If some of these classes should not be associated with element types, because they do not have local behaviour and should not be part of any coordination, you can safely delete them[3]. In the same way, you can delete ECNO references and inheritance relations between element types from the diagram, if they are not relevant for the behaviour. Before you continue with editing the model and adding coordinations, make sure that you give the ECNO model a unique URI, which the ECNO engine will use for loading the plugged in packages dynamically. These URIs do not play a role as long as you run single ECNO packages as *Java applications*; once you start running ECNO packages as *Eclipse applications*, these URIs are very important, since models are plugged in to Eclipse with this URI and loaded dynamically by the ECNO engine by referring to this URI.

5.1.2.2 Adding event types

Note that, initially, there will not be any event types in the coordination diagrams because the Ecore model does not give any clue which event types there should be. This is completely up to ECNO coordination diagrams. Event types can be easily added by using the ECNO Diagram editor's respective tool and, then, placing the event type somewhere on the canvas. You also need to give the event type a name that is unique within the diagram resp. package.

You can also add parameter to an event type. To this end, select the respective tool in the editor and, then, click in the parameter compartment of the respective event type. You can add the name directly by clicking on the parameter; the parameters type must be set by selecting the parameter, and then changing the value of EType in the properties view. The type can be selected from a drop down menu, which shows all classes and data types of the underlying Ecore model as well as standard data types of Ecore. If you want to select an other data type, you need to load the respective resource by the Editors standard "Load Resource..." mechanism.

In the properties view, you can specify whether the parameter should be a collective parameter or not. By default, a parameter will be not collective (exclusive). If you set the collective property of a parameter to true in the properties view, the respective parameter will be decorated with a trailing

[3]If you should realize that you need to make them element types later, the editor provides you with means for adding them again, which is discussed in Sect. 5.1.2.4.

start, in order to show in the diagram that the parameter is collective. Note that the other properties do not have any meaning in ECNO; they are inherited from the Ecore meta model, since ECNO re-uses these concepts as far as possible (see Sect. 5.6), but do not have a meaning for event parameters of ECNO.

5.1.2.3 Adding coordinations

Once you have added some event types, you can start adding coordinations. To this end, you would first create some coordination sets for element types by using the respective tool of the editor. Note that each coordination set needs to be associated with an event type, its trigger event. This is done in the properties view, where you can select one of the event types that are defined in this diagram. Note that you must select an event type from the diagram itself. You can also change the coordination sets priority.

Then, you can start adding coordination annotations. Note that the respective tool is called "synchronisation", since it actually associates a reference with a coordination set. Such a synchronisation is created by selecting the respective tool, and then dragging the mouse from a reference to the respective coordination set. Note that the synchronisation must run between a coordination set of an element type and a reference which is owned by that element type[4]. Moreover, the synchronisation needs to be explicitly associated with an event type; typically, it is the same as the trigger event type of the coordination set. When you have selected a synchronisation, you can also change its quantifier between "ONE" and "ALL".

5.1.2.4 Adding element types and references

As discussed in Sect. 5.1.2.1, the coordination diagram editor automatically creates element types for each of the classes of the underlying Ecore models, when you select the underlying Ecore model (for a still empty coordination diagram). And typically you would delete some of these element types and references only.

At some point later, it might happen that you would like to turn a class into an element type or that you would like to add a coordination annotation to a reference that you had deleted in the initial setup. In order to do that, the coordination diagram allows you to add element types and references to the coordination diagram explicitly: just use the respective tools from the tool palette of the editor for this purpose. The newly created element types and also the references will, however, not be automatically be attached to some class or reference of the underlying Ecore model. You need to attach the element type or the reference to the underlying class or reference in the Ecore model manually: this is done by selecting the element type or

[4]This constraint is not yet checked by the editor.

reference, and then changing the EClass or EReference attribute to the one it should be. Note that the editor does not validate the correctness of a Ecore reference that you choose for a ECNO reference, which must be a reference between the underlying classes of the respective element types. Therefore, you need to make sure that this is the case yourself[5].

Note that you cannot change the name of an element type or the name of the reference in the coordination diagram editor. The name of an element type is derived from the name of the underlying class in the Ecore model.

5.1.2.5 Subtyping of references

Note that you are allowed to add a reference between two element types so that the target element type of the ECNO reference, actually refers to a subtype of the target class of the corresponding reference in the Ecore model. We have seen two examples of such references in the coordination diagram for Petri nets in Fig. 2.4 on page 33: the references source and target in the coordination diagram refer to the Place element type, whereas these references refer to the class Node in the underlying Ecore model.

We call such a use of an ECNO reference *sub-typing* of a reference, since it does not refer to all the corresponding links of the Ecore model, but just to the ones that end at objects of the respective type (Place in our example). And the coordination annotations refer to the links going to elements of this type only. In order to warn you about this, the coordination diagram shows such sub-typed references in parentheses.

5.1.2.6 Importing types

In the example in Sect. 4.1, we have seen that an ECNO model can be built up from different packages. In order to establish relations between different packages, it is necessary to import element types of one package to another one. ECNO provides a very general mechanism for importing element and event types from other ECNO packages, even if they use different underlying object-oriented technologies. But, it is easiest to import types from ECNO models that have underlying Ecore models, and we explain this mechanism here.

In order to import an element type form another package, you would first create an element type with the element type creation tool (as discussed in Sect. 5.1.2.4). Then you would select this new element type and, in that properties view, select for the "Import" property the element type, which should be imported. Note that it might be necessary that you load the respective ECNO model as a resource as discussed in Sect. 5.1.2.1. Note that

[5] Eventually, there will be a validation feature in the coordination diagram editor for this purpose.

an imported element type will get a decoration, which graphically indicates that the element type is imported.

Importing event types, basically, works in the same way: you create a new event type, set a reference to the event type that is imported, and give the imported event type a name, which it is referred to in this new package. Note that all references to these event type – including the use in the respective ECNO nets – needs to refer to this type name, not the name withing the original package from which it is imported.

After setting these imports, you need to do a last step: Right-click into the canvas of the coordination diagram and select "ECNO→Resolve imported elements and types". This will set the attributes "Package URI" and "Type Name" for all the imported element and event types. And this is the actual information used by the ECNO engine for resolving cross references between different packages. Only after resolving the imported types, the names of the element types will properly show in the diagram. The "Import" property is just a matter of convenience so that you do not need to remember and explicitly edit long package URIs manually. You can just select the respective element or event type that is imported.

Note that an imported event type can actually be given a name that is different from the the event type type that is imported. This allows you to avoid name clashes of event types that you define in your package with names that are imported. Note that all references to these event type within this package – including the use in the respective ECNO nets for the local behaviour – must be via this name given to the imported event type; the name of the event type in the original package does not have any meaning within this package. But, as long as there are no name clashes, you can of course give the same name to an imported event type as the one it had in the package from which it was imported.

5.1.3 ECNO nets

In this section, we discuss how to create and edit ECNO nets. The ECNO net editor is implemented as an extension of the ePNK [31]. Therefore, creating and editing ECNO nets follows the general principles and steps of the ePNK, which are discussed in Chapter 3 "User's guide" of the ePNK user's and developer's guide [35]. Therefore, we give a rough overview of the main steps only with the focus on the specific steps and the extensions that are relevant for ECNO nets.

The ePNK, uses the *Petri Net Markup Language* (*PNML*) [23, 20] as format for all kinds of Petri nets. A single PNML file can contain several Petri nets. For a single ECNO package, all ECNO nets for the local behaviour of elements for that package must be contained in a single *PNML document*. For creating a new PNML document, you can use the wizard for PNML documents ("New→Other..."), which you can find in category

"ePNK"; using "PNML" in the filter in the "Select a wizard" dialog should direct you to the PNML Documents wizard. This will create a file with extension ".pnml". When you create the PNML document, the PNML editor will open right away. Later, you can use the PNML Editor (default editor for PNML files) for opening PNML files.

The *PNML Editor* is a tree-based editor. When newly created, the document contains an empty document (top-level element) only. When you right-click on this document (Petri Net Doc) and select "New Child", you will have the option "Petri Net (`http://se.imm.dtu.dk/ecnonet`)", which will create a new ECNO net. On this newly created ECNO net, you should create a name ("New Child→Name"), which you then can edit in the properties window. Note that the name should be the same as the corresponding Element in the ECNO coordination diagram. On the ECNO net, you should also create a new page ("New Child→Page").

By double-clicking on that page (alternatively you can select the page and select "ePNK→Start GMF Editor on Page") you can open the graphical editor of the ePNK on that page[6]. In this graphical editor, you can then draw the graphical structure of the ECNO net by using the tools for places, transitions and arcs. Note that the tree editor will still remain open; and you need to save the net from the tab of the tree editor.

Adding names for places and transitions and event bindings, conditions, and actions for transitions is done by using labels. You would first create a "Label"; then you would choose the "Link label" tool and drag it from the new label to the resp. place or transition; then you will be prompted to choose the kind of label it is supposed to be (dependent on the still available options). Then, by clicking on the label you can edit it. If for some reason, you forgot which label was of which kind, you can click on it; then the properties view will show you which kind of label it is.

Note that you can also add some import and attribute labels to a page of an ECNO net. To this end, you would select the "Page Label" tool. Once you click to the position, where the label should be, you will be asked which kind of label it should be (import or attribute). If you select an attribute, make sure to make constant attributes (Java keyword final), since non-constant attributes would compromise saving and loading the state of an ECNO application.

In the end, you should select the tab for the tree editor of the PNML document again and save the file. It would be a good idea to double-click on the top-level element "Petri Net Doc", which will add unique identifiers to all elements of your net[7]. Note that, in the same top-level element, you can

[6]The first time, you open the graphical editor you will be asked to confirm that you are sure; the reason is that opening the graphical editor for the first time will prevent to undo this and all previous actions.

[7]This is not strictly necessary, but it is recommended since PNML documents without these unique ids are not conformant to ISO/IEC 15909-2.

– actually you must – add all the other ECNO nets for the package. This works as discussed above.

Note that you can split up a single ECNO net into different pages, if you have larger ECNO nets for the local behaviour of the elements. Splitting up larger nets follows the principle of the PNML and the ePNK using pages and reference nodes. But we do not discuss the details here. Typically, ECNO nets should be small enough to fit to a single page.

5.1.4 Creating instances

In order to create an ECNO application, we need to create an actual *instance* of the Ecore model, which defines the *start configuration* of the ECNO application. There are two different ways to create an instance, both of which are independent of ECNO; they are EMF concepts.

The first way of creating an instance of an Ecore model is creating a so-called *dynamic instance*. In order to create a dynamic instance, open an Ecore model with the tree editor and select the class, which should serve as the container for all the other objects – in our examples, this would be the classes Setting, the class Petrinet, or the class VendingMachine. When you have selected the container class, right-click and select "Create Dynamic Instance..." from the pop-up menu. In the opened "Dynamic Model" dialog, select a folder and a name for the file (make sure to keep the proposed extension ".xmi") to which this dynamic instance should be saved and, then, finish the dialog. Then, a tree editor with an instance of the container class will open, and you can add new child elements to this container, as usual for the EMF tree editors; you can also change attributes and references between different elements in the properties view. Also saving the file works as usual for EMF tree editors. If you want to open such an file later, you can open it with the "Sample Reflective Ecore Model Editor...". Note that you can create these dynamic instances without having generated code for the editor from the Ecore model. Therefore, dynamic instances would be used for creating instances when you want to run ECNO applications as Java applications.

The second way, of creating an instance of an Ecore model would be generating the code for the EMF tree editor (or a GMF editor) and starting the Eclipse runtime workbench and using the generated editor in this runtime workbench for creating and editing the instance. This is done as usual in EMF, and is not discussed here. But, we will briefly come back to this when we discuss how to set up and run an ECNO application as an Eclipse application in Sect. 5.4.1.

5.2 Code generation

When we have all Ecore and ECNO models as well as the start configuration of the ECNO application in place, we can start generating the code. Generating the code, requires some extra information such as qualified package names as well as a name for the class that represents the ECNO package. For EMF, this additional information comes from the so-called EMF *gen model*. For ECNO, there are similar gen models.

In this section, we discuss how to create these gen models and the meaning of the different properties of these gen models. Moreover, we discuss how to generate the Ecore and the ECNO code from these gen models and how the generated code looks like.

5.2.1 Ecore model code

As discussed in Sect. 5.1.1, before any ECNO code is generated, the code for the Ecore model should be generated.

The steps for creating the EMF gen model and for generating the code for the Ecore model follow the standard steps of EMF [8]. Therefore, we go through these steps quickly. For an Ecore model, an Ecore gen model can be generated by selecting the file containing the Ecore model, and then selecting "New→Other...", and in the then opened "Select a wizard" dialog, selecting the "EMF Generator Model" wizard, and following through this dialog. In principle, you would not need to change anything in this generator model; but, it might be a good idea to change at least the base package property of the EMF package, so that the generated Java packages are not placed at the top-level.

From the open EMF gen model, you can generated the Java code for the model code, the edit code, and the editor code by a pop-up menu. If you want to run your ECNO application as a Java application only, it is actually enough, if you create the model code – you will not need the edit and editor code. In case you want to run the ECNO application as an Eclipse application, you need to generate the edit code and the editor code too. We come back to that in Sect. 5.4.

5.2.2 ECNO model code

Next, we discuss how to generate the code from the ECNO models, the coordination diagram as well as the ECNO nets. Similar to EMF, some more technical information such as qualified package names, class names, and some additional information for configuring the code generation from ECNO models need to provided in an *ECNO gen model*. The ECNO gen model, in addition, combines a coordination diagram with the PNML document containing the ECNO nets with the local behaviour for the element types of

the coordination diagram. Together, a coordination diagram and the PNML document for the ECNO nets represent an ECNO package, which in turn is based on an Ecore or EMF package.

5.2.2.1 ECNO gen model

Let us briefly discuss how to create and edit an ECNO gen model and what the different properties mean. Actually, an ECNO gen model contains a single object only, which has several attributes.

An ECNO gen model can be created by a wizard ("New→Others...") which is called "Ecnogen Model" and there is a corresponding EMF tree-editor for ECNO gen models. The attributes of the ECNO gen models are the following:

Ecno Model The reference to the ECNO coordination model.

Behaviour Model The reference to the PNML Document that contains all the ECNO nets with the local behaviour of each element type (note that element types for which no behaviour is defined will have a simple default behaviour). The Petri nets in this PNML document should all be ECNO nets, and the name of the net should be the name of the element type it is defining the local behaviour for.

Emf Gen Model The reference to the EMF Generator model, from which the model from the Ecore model was generated; it is possible to provide a reference to several EMF Gen models here, in case different ECNO packages and Ecore packages are used. The first reference, however, should be the one to the EMF generator model for the Ecore model underlying the ECNO coordination model above.

Model Class Name The name of the generated class that represents the ECNO coordination model.

Automata Factory Class Name The name of the factory class, which will be created by the code generator. This factory class is used by the ECNO engine for creating the local behaviour for new elements that have an element type that is defined in this package.

PackageAdapter Factory Class Name (optional) If the ECNO model should be registered as an extension with Eclipse, this is the name of the respective factory class. This is not relevant, if the generated code is run as Java application only; it should be set, however, when the ECNO model should be run as an Eclipse application, which is discussed in Sect. 5.4.

Base Path ECNO Automata The base package (path) to which all the code for the automata for the local behaviour is generated.

Base Path ECNO Events The base package (path) to which the code for the events (actually the event values class) is generated. These classes provide the access to the values of the involved events in event bindings, in conditions, and in actions at runtime.

Base Path Model Class The base package (path) to which the class representing the coordination diagram is generated.

Required (optional) An ECNO model may refer to other ECNO models. In order for the code generator to refer to the generated code for these projects, the code generator needs to know the ECNO gen models for these ECNO models. The attribute required refers to the ECNO gen models of packages that are referenced in ECNO model. The name "required" has historic reasons.

Note that before you can set or add a reference to other models (PNML, ECNO coordination diagram, EMF gen model, etc.) you need to load that resource to the editor. You can do that by a right-click and then selecting "Load Resource..." and then selecting the resp. resource from the workspace.

5.2.2.2 The Generated model code

Once you have created the ECNO gen model, generating the code for the ECNO package is easy. In the ECNO gen model, right-click on the gen model element, and select "ECNO→Generate ECNO Package Code". This will start several Eclipse jobs that run in the background. Should you get some error messages indicating that some classes could not be found, it might help to delete the project `.JETEmitters` from your workspace (you will find it if you open the "Navigator").

Here, we do not discuss the details of the generated code. But, we give a brief overview of different packages and classes that are generated and their purpose. In addition to the Java packages created by the EMF code generator, the ECNO code generator creates four different Java packages, which correspond to the three paths given in the ECNO gen model:

Local behaviour package In this package, there is a class for each ECNO net, which represents the local behaviour for each element type of the package. The names of these classes are the same as the corresponding element type[8]. In addition, there is a class representing a so-called EMFBehaviourAdapter, which knows for which EMF classes there exists a local behaviour, and which provides methods for creating new instances of the local behaviour, when the ECNO engine encounters new objects, which correspond to an element type. This class is used

[8]Since the names are the same, the qualified package name must be different from the corresponding EMF model package.

by the `EMFPackageAdapter`, which is also generated by the ECNO code generated. This PackageAdapter will be registered with an Engine running this package.

Package adapter package This package contains the class package adapter, which we had mentioned above. The `PackageAdapter`, is a programmatic description of the underlying ECNO coordination diagram. It knows, which element types there are, which local behaviour these types have, as well as the event types and the coordinations between the element types. These `PackageAdapters` are registered with the ECNO engine and will be used by the ECNO engine to compute the possible interactions, and to initialize the local behaviour of newly encountered element instances.

If the ECNO gen model is configured to plugin the ECNO package as an extension to Eclipse, the package adapter package contains another class which is called a `PackageAdapterFacory`. This is actually the case, if the name for the `PackageAdaptorFactory` in the ECNO gen model is not empty. And the name given there will be the name of the generated class for the `PackageAdaptorFactory`. This `PackageAdaptorFactory` is used by the ECNO engine to create a new `PackageAdapter` for this package, when the ECNO application is running as an Eclipse application and configured to use this package (which is discussed in Sect. 5.4).

Events package For each event type and event type extension that define at least one parameter, the ECNO code generator generates a class that represent the values of the parameters of an instance of the event. These so-called *event value* classes are generated in this package. These classes have the same name as the respective event types and event type extensions – but staring with a capital letter.

In addition to generating the classes that we discussed above, the ECNO code generator will also configure the project, if the ECNO gen model is configured to create a `PackageAdaptorFactory`. In that case, the packages for the events and for the package adapter are exported (added to the "MANIFEST.MF" file of the Eclipse project) and an extension referring to the `PackageAdaptorFactory` is added to the "plugin.xml" file of the project.

5.2.3 ECNO as Java applications

As discussed above, there are two ways of running the code generated from ECNO models. The simple way is running the code as *Java application*. When run as a Java application, the application always starts in the same state, which is defined as an instance of an EMF model as discussed in

Sect. 5.1.4. If you want to save a state of an application, you need to run it as Eclipse application, which is discussed in Sect. 5.4.

Starting an ECNO model as a Java application requires to generate code for the instance to be started. In this section, we discuss how to do this. Similar to the ECNO gen model, we need to create an ECNO instance gen model first, from which the code for the Java application will be generated. The ECNO instance gen model can be created by the "Ecnoinstancegen Model" wizard. Like for the ECNO gen model, the ECNO instance gen model contains one element only, which can be edited with a simple EMF tree-editor. The ECNO Instance Gen Model element has four attributes:

Instance This refers to initial configuration, which is an instance of an EMF model as discussed in Sect. 5.1.4.

Instance Class Name This is the name of the class, which represents the initial configuration, and which is the Java application (with a static main method so that it can be started as Java application).

Base Path ECNO Instance This is the qualified name of the package to which the Java application is generated.

Ecno Gen model This is the ECNO gen model of the ECNO model, which should be used for the ECNO application (indirectly, this is referring to the actual ECNO models).

From this ECNO instance gen model, the code for the Java application can be generated: "ECNO→Generate ECNO Instance Code". This will generate one class, which represents the initial configuration of the ECNO application; running this Java class as Java application will start the ECNO application.

Note that the generated Java application will always start the ECNO application with the simple default GUI (see Sect. 5.3). If you want to change that, you would need to change the code manually. Since Java applications, are more for experimentation, there is no way to configure the code generation to start with own GUIs. This is possible for ECNO applications that run as Eclipse applications, which would be used in more realistic applications anyway.

5.3 ECNO GUI

The default GUI for ECNO applications mainly serves the purpose of quickly getting some application running and playing around with it. For all explicitly registered elements (the ones added to the engine by the `addElement()` method) the default GUI will show a panel, which for every GUI event type of that element type will show a button. The button for an event type will

be enabled, when there is an interaction for that element with the respective event type enabled. Otherwise, the button will be disabled. Pressing the button will execute the enabled interaction.

Note that this default GUI is not supposed to be the GUI of final applications. In Sect. 5.5.2, we will discuss how to implement customized GUIs based on ECNOs Element Event Controllers.

Since the main purpose of default ECNO GUI is playing and experimenting with ECNO applications, it provides some additional features. First of all, when hoovering over an enabled button, a tool tip will open which shows the underlying enabled interaction: the participating elements with the events they are involved in, and for each event, the parameters are given. This might help, to understand for an ECNO application which interactions are enabled.

Another important feature of the default GUI is that it shows when more than one interaction for an element and event type are possible. This will be indicated by an asterisk (*) decorating the respective button. Then the user can iterate through the different possible interactions for that element and event type, by clicking the mouse button in combination with the "SHIFT" key; when the last possible interaction is reached, the decoration will change to an exclamation mark. By pressing a button together with the "CONTROL" key, the button can be re-initialized so that the iteration through all interactions starts from the first possibility again.

Together with the tool tip indicating the possible interaction, this provides a simple tool for seeing all the available interactions. In ECNO applications that run as Eclipse applications, the user might even undo and redo interactions (see Sect. 5.4), which should make it even easier to analyse the behaviour of an ECNO application.

5.4 ECNO Eclipse application

For more realistic ECNO applications, we would not want to start them in the same state every time. Rather, we would like to save the state in which they are, when we shut them down; and when we start them again, we would like to start them in the state in which they were shut down. This functionality is not available when ECNO applications are run as Java applications. But, when we run ECNO applications as Eclipse applications, we can save the state of an ECNO application and start the application in that state again. In addition, there are some other features such as undo and redo and some statistics of computation times available when an ECNO application is run as an Eclipse application. All this additional functionality will be available via the "ECNO Engine registry" view, which gives an overview of all running ECNO engines and a way to control these engines. We explain the "ECNO Engine registry" in Sect. 5.4.2.

5.4.1 Setting up a configuration

Before discussing the "ECNO Engine registry", we need to explain how to create a *start configuration* for an ECNO application. Similar to an ECNO application that runs as a Java application, the start configuration needs an instance model, which defines the elements that are initially there and how they are related. In addition, the start configuration needs to define which ECNO packages it should use, and the start configuration can define which GUI and which controllers the ECNO engine should used.

We use the Petri net example (project `APetriNetEditorIn15Minutes.runtime`) from Sect. 2.1.2 again to discuss the main concepts of configurations. We use the configuration "a_net.behaviourstates" from the "run" folder of that project as an example. The extension ".behaviourstates" results from the main purpose of this file, which we will explain in more detail later: it is used to save the states of the local behaviour of each element. The reason is that normal EMF instance files can only save the information that is represented in the Ecore model; therefore, the states of the local behaviour need to be saved somewhere else: in this configuration file. We will discuss some details later.

For setting up a start configuration, we need only a few concepts of all the features of "behaviour states" files. The top-level element, "Behaviour States" is a container which contains the behaviour states of some elements. The "Behaviour States" element has a single[9] property "Packages", which refers to the unique identifiers of all the ECNO packages that this ECNO application is using. Note that the ECNO packages that these unique identifiers refer to must be plugged in to Eclipse for the ECNO engine to run properly from that start configuration.

The "Behaviour States" element can contain several elements "Behaviour State". Each "Behaviour State" element that is contained in the top-level element, refers to an element (which typically will be a reference to an element in some instance file) and it contains the state of the local behaviour of that element. The implementation of the state depends on the used technology and the implementation for the local behaviour. In our case, it will be "PT Net States", which represent the current marking of the ECNO net. For easing the setup of a start configuration, there are some special states implemented. The first is called "Default State", which would initialize the local behaviour of the respective element type in its default state as defined in the local behaviour. For an ECNO Net, this would be the initial marking. For large start configurations, however, it would still be very tedious to set up such a start configuration manually. Therefore, there is another special stated called "Default Container". This will initialize all

[9]There is one other attribute "Added", which defines which elements are currently added to the ECNO engine. This however, is more relevant for saving the state of an ECNO application than for defining the start configuration.

the elements that are contained – directly or indirectly – in this element with their default states. In our Petri net example, there is only a single "Element Behaviour State" in the configuration, which refers to the Petri net itself, and the element's behaviour is "Default Container"; this has the effect that the container as well as all the elements contained in it will be initialized in their initial state. Note that this, on the side, will have the effect that the element that are GUI element types will also be added to the ECNO Engine (and this way to the GUI). Therefore, there is typically no need to "Add" elements to the top-level element of the configuration explicitly.

The top-level "Behaviour States" element can contain one other kind of element, which is called "Controller Configurator". The "Controller Configurator" refers to the URI of a plugged in Controller Configurator, which will take care of setting up the GUI for this ECNO application and some other Engine Controllers. We discuss the details of how to program and to plug in such Controller Configurators in Sect. 5.5.2.5. For now, it should be sufficient that they set up the GUI. If there is no such configurator, the ECNO Engine will start up with the standard GUI. The Controller Configurators might also have some state information, which they need in order to properly start up and to save the current state of the controller. To this end, the "Controller Configurator" can refer to some object. In the case of our Petri net example, this object is the Petri net diagram, since the Controller Configurator that we use here configures the GUI in such a way that it shows the current state of the Petri net in the GMF editor.

As discussed above, the states of an element refer to an element. These elements and the relation between them would typically be stored in a different file, which is an instance of some EMF model. Since we are now running ECNO as an Eclipse application, the EMF code and the code generated from the ECNO packages would be plugged in to Eclipse or we would run Eclipse as a runtime workbench. Therefore, we can use the Eclipse tree editor or a generated GMF editor for creating and editing these instances. We would not need to use the editor for dynamic instances of Ecore models any more.

A new configuration file can be created by "New→Other..." and then selecting "Behaviourstates Model". The reference to the instance model would be created in the "Behaviour State" elements via the "Element" property. In order to set this reference, we would need to add the file with the instance model by loading it as a resource to the EMF tree editor as usual.

This should be enough information for setting up a first start configuration for an ECNO application. We will discuss below how to start and control an ECNO application from there. When we start an application from such a configuration file, and the state of the application is saved later, this will be saved in the same file it was started from. This will add much more information to this file, which represents the current state for each element individually, and orphaned elements (elements that are not contained in any

resource) will be added to this file too. We do not discuss the details here. Since saving the state of an ECNO application will overwrite the original start configuration, you would probably like to make copies of the configuration file and all the instance files it is referring to in a different directory, so that you do not loose the original start configuration. Eventually, the ECNO Tool might also provide a means for automatically setting up a new start configuration file from some configuration information. For now, you need to do that manually – and, in order not to loose that information, you should save it together with the instance in a separate folder.

5.4.2 Running a configuration

Once your have a start configuration, you can start an ECNO application from it – assuming that all the ECNO packages the top-level element is referring to are properly plugged in to Eclipse.

In order to start an ECNO application, you would select the file with the start configuration in an Eclipse resource browser, right-click on it and select "ECNO→Start ECNO Engine". Note that you should make sure that the configuration itself as well as any of the instances it is referring to are not open in any editor at that time.

We had seen an example of an ECNO application running as an Eclipse application in Fig. 2.2 on page 30 in Sect. 2.1.2 already. Note that, in that example, the graphical editor for the instance is open; the reason is that it is opened as part of the GUI of that ECNO application. But, this editor should not be open before the ECNO application is started.

You can start any number of ECNO Engines at the same time, but you should not start it from the same configuration or instance (since the state of the two different ECNO applications will be saved to the same files and you will loose information of one of the running applications). The ECNO Engines can be controlled via the "ECNO: Engine registry" view, which can be opened via Eclipse's "Window" menu ("Show View→Other..."). Figure 2.2 on page 30 shows a screenshot of an ECNO application running as an Eclipse application, and the view at the button shows the "ECNO: Engine registry". The currently running ECNO applications are shown in rows, and the rows can be selected by clicking on them. The buttons for the "ECNO: Engine registry" view refer to the selected row – except for the "Delete" button, which refers to the checked (checkboxes in the front of every row). The "Delete" button will shut down all checked ECNO Engines. For each row resp. running engine, the "Engine name" can be freely chosen and changed by the user; the "Resouce name/path" shows the configuration from which the ECNO engine was started – and to which the state is saved, when the "Save" button is pressed.

For a selected ECNO engine, the "Back" and "Forward" button will undo and redo the last executed interaction of the engine. This is mostly

relevant for experimenting with ECNO applications; using them can actually be a bit tricky, once interactions are executed automatically in a quick succession by some automatic controllers (see Sect. 5.5.3). The "Rubber" button between the "Back" and "Forward" button can be used to clear the history of executed interactions, which will, of course, prevent to undo and redo them.

Further to the right, there is a "Save" button, which will save the current state of the ECNO application. If the state of the ECNO application is saved, it will start from exactly this configuration the next time it is started from the configuration. Remember that ECNO is not fixed to using EMF as underlying technology for the structural models; the implementation of the undo and save mechanism, however, is dependent on the underlying technology; also the save mechanism depends on whether the underlying technology supports it. Therefore, the undo/redo mechanism as well as the save mechanism might be disabled, if the underlying technology supports them. For all the examples of this report, however, the undo/redo and save mechanisms work since they are all based on the EMF technology.

The last button is the "Information" button. This will open a dialog with some statistical information on the run ECNO application and the currently registered elements. Note that elements of an ECNO application will be garbage collected automatically, when they are no longer in use. But, the undo/redo history might keep elements from being garbage collected; so, if you want to have a more accurate picture of the current situation, you should clear the history of interactions by pressing the "Rubber" button[10].

5.5 Programming with the ECNO Framework

As we have seen in Sect. 5.2, we can generate a complete application from ECNO models. By default, this application will run with a simple ECNO GUI, which was discussed in Sect. 5.3. This GUI can be used for testing an ECNO model and for experimenting with ECNO; for real applications, however, this GUI is not good enough. Therefore, the *ECNO programming framework* provides means to program customized GUIs.

Though, programming a customized GUI is probably the most important use of the ECNO programming framework right now, the ECNO programming framework also provides mechanisms for computing and executing interactions programmatically (part of which will be used when programming GUIs), and it provides mechanisms for programming the local behaviour of elements. Though we would not advocate programming the local behaviour

[10]Note that customized GUIs might also be responsible for keeping references to outdated elements which are no longer referenced in the ECNO engine itself. This happens for example, if in our Petri net example, where the graphical editor for Petri nets, which is part of the GUI to show the token game, keeps references to all elements. This is why, in the Petri nets example, the statistics of current elements is not accurate.

of elements manually, this interface can be used to generate code for local behaviour from other models than ECNO nets. Actually, it is even possible to programme class diagrams and the coordination on top of them manually.

In this section, we discuss how to compute and execute legal interactions of an ECNO application (Sect. 5.5.1), and how to program GUIs on top of these mechanisms (Sect. 5.5.2). In addition, we discuss how to change the elements of an ECNO application and their relation programmatically outside interactions without compromising ECNO's mechanism for transactionality (Sect. 5.5.3); and we discuss how to automatically execute interactions when the become enabled. In addition, we give a brief overview on some advanced features (Sect. 5.5.4), which will not be discussed in detail.

5.5.1 Computing and executing interactions

We start with explaining how to compute and execute the interactions of an ECNO application programmatically. To this end, we assume that we have access to the instance of the ECNO `engine` which is running the ECNO application, and some `element` and some `eventType`. We will see in Sect. 5.5.2 how this information would be typically passed on to *element controllers* which are automatically installed by *Engine Controllers* and how types can be looked up in the *ECNO package adapter* (see Sect. 5.6 for some more details). For now, we just assume that we have this information available.

A call `engine.getInteractions(element, eventType)` computes all the possible interactions for the given element that involve an event with the respective type. Actually, the result is a Java iterator, which would return all possible interaction one after each other. This iterator is called `InteractionIterator`. With the usual operations such as `hasNext()` and `next()`, we can find out whether there are possible interactions and obtain the next `Interaction`. Once we get hold of an interaction, we can execute it by calling its `execute()` method.

Note that the ECNO engine and the execution of interactions is implemented in such a way that it is thread-safe, even when interactions are computed and executed concurrently in different threads. In particular, interactions will be executed only if the interaction is still valid at the time when it is executed, and if it is executed it will be executed *atomically* (completely) and in *isolation* (without the execution of other interactions interfering). And when run as Eclipse application with Ecore models, the ECNO engine also guarantees *consistency* after the execution of an interaction (only if the execution of the interaction does not violate any of the models' constraints, the interaction is executed – otherwise the interaction will be rolled back). This guarantees transactionality respecting the A, C, and I of the *ACID-principle* [13] – the "D" for durability is not yet fully supported, though; there is only a very basic mechanism for explicitly saving the state of an ECNO application.

Since interactions can be executed from different threads, it can well happen that an interaction that was returned by the interaction iterator will become invalid before its is actually executed. There are three mechanisms that help dealing with this situation. First, the interaction has an `isValid()` method, which returns true, if the interaction is still valid at the time when the method `isValid()` is called. It is a good idea to call this method before actually executing an interaction – in particular when some time has elapsed since the interaction was computed. Second, we can register some *invalidation listeners* with an interaction and even with an interaction iterator, which is called when an interaction becomes invalid or when the interaction iterator becomes invalid, which means that one of its interactions becomes invalid. This mechanism is discussed in Sect. 5.5.2 since this is at the core of building GUIs for ECNO. Both of the above mechanisms, however, are not enough to make sure that the `execute()` method is invoked on valid interactions only in a multi-threaded setting: It can happen that the interaction becomes invalid, just when the `execute()` method is called. If this happens, the `execute()` method will raise an `InvalidStateException`. If this happens, the interaction will not have executed – not even partially (atomicity). The `InvalidStateException` is a runtime exception. Therefore, applications are not required to catch it. But, at least in a multi-threaded setting, it is highly recommended to catch these exceptions and properly react to it. Note that also the `next()` and `hasNext()` methods of the interaction iterator might throw an exception `InvalidStateException`, when the iterator becomes invalid.

With calling `engine.getInteractions(element, eventType)` all interactions for the given element in which at least one event of the given event type is involved are computed. Sometimes, we would like to compute interactions in which the event has some parameters set to some specific values. To this end, the engine can be called with a specific even with this parameter set: `engine.getInteractions(element, event)`. We will discuss how to create an event of a specific type and how to set its parameters in Sect. 5.5.2 below.

As long as all the changes of an application are made by executing interactions only, the ECNO engine guarantees the transactionality of their execution. In some applications, we might want to make some changes on the elements and their links by programming these changes using the API of the underlying Ecore or object-oriented model. In order to maintain transactionallity, the elements on which the changes are made programmatically need to be locked explicitly. This is discussed in Sect. 5.5.3 where we explain some issues of ECNO that concern multi-threaded programming.

5.5.2 Customized controllers and GUIs

In our workers example, which we had used in the first informal introduction to ECNO in Sect. 1.1, we had discussed the simple default GUI of ECNO applications in Figures 1.10–1.12 on page 21. This default GUI for ECNO is good enough for experimenting with ECNO. For more realistic ECNO applications, however, we would need more; the GUI should look more fancy, and there should be other ways to interact with the application than pressing buttons.

In this section, we show how to equip ECNO applications with more advanced GUIs. To this end, we discuss how to implement a GUI for the workers example that resembles a work list in workflow management. Figure 5.1 shows how this GUI looks like: in particular, it shows the possible actions from a single worker's perspective, which shows all possible jobs the worker could be participating in right now; moreover, it allows the worker to enter some text, which is used in the ECNO application (as the name for the newly created job).

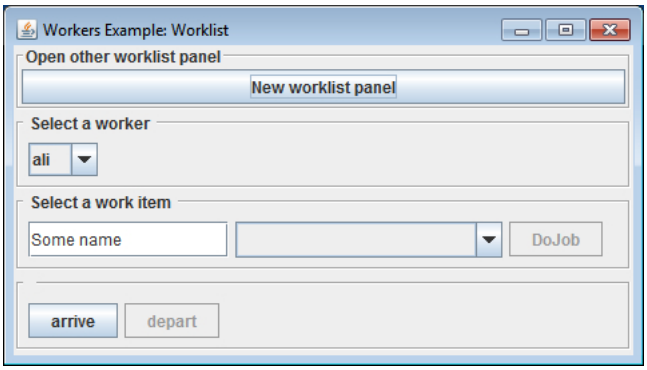

Figure 5.1: Worklist GUI for workers example

5.5.2.1 The workers example and its GUI

Before explaining how to programme this GUI and the underlying concepts of ECNO, let us briefly discuss the GUI from Fig. 5.1. Remember that the workers example was about modelling workers that arrive and depart from work together; when at work, there are jobs that they can do. Each of the jobs requires different workers to participate in the job when it is executed. In addition to the example from Sect. 1.1, we slightly extended the example so that jobs have a name. Similar to the model from Sect. 1.1, the actual "work" of the job consists of creating the a new job that requires the same workers as the job itself; in our extended version, the new job gets a new name; this name is provided as a parameter of the event, which is provided by the worker initiating the job via the text field in his GUI (work list). In

order to make the example slightly more interesting, we also add a condition that this name must have at least six characters (in the ECNO model, this condition is a part of the local behaviour of the job).

The GUI shown in Fig. 5.1 consist of four main parts, which we call panels. The top-most panel allows the user to create another instance of the GUI attached to the same application; this way, it is possible to see several work lists for different workers at the same time. Note that, once started, these different GUIs are completely independent of each other and are able to show the state of the workers application from the perspective of the chosen worker. The "Select a worker" panel allows to select the worker for which the possible jobs and possible actions should be shown. The possible jobs (work items) for the selected worker are shown in the "Select a work item" panel. This panel consists of three parts: a text input field in which the name of the new job to be created can be entered, a drop down menu from which one of the currently possible jobs can be selected; in the example from Fig. 5.1, this drop down menu is empty, since the chosen worker "ali" has not yet arrived at work. The last panel shows the "arrive" and "depart" actions, which indicate that the worker arrives or departs from work (remember that the workers "ali" and "bert" as well as the workers "cleo" and "dan" share a car and, therefore, arrive and depart together).

Figure 5.2 shows the same worklist GUI after all workers have arrived and "dan" was selected as the worker. It shows that "dan" can take part in three different jobs – shown in the drop down menu. Pressing the "doJob" button would execute the selected job, which creates a new job with the name "Some name" – the value currently provided in the text input field. Note that, if we entered a name shorter than six characters to the text input field, the drop down menu would be empty and the "doJob" button would not be enabled since the ECNO model requires the name of the new job to be at least six characters long.

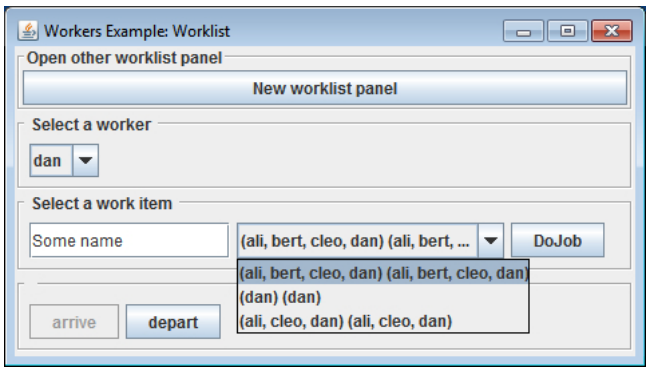

Figure 5.2: Worklist GUI after all workers arrived

We do not discuss the details of the extended ECNO models for the

Listing 5.1: Use of `ElementButtonPanel`

```
workerPanel = new ElementButtonPanel(engine, null, selected);
this.add(workerPanel);
this.setVisible(true);
```

workers example here; as compared to the models discussed in Sect. 1.1, there are only a few minor changes. If you want to have a look at the extended models, you will find them in the project `dk.dtu.imm.se.ecno.example.workers.worklistgui`, which is deployed as one of the example projects[11] together with ECNO. You can start this application by starting the class `SettingWorkersGUI` in the package `dk.dtu.imm.se.ecno.examples.workers.instances` as a Java application.

5.5.2.2 Using GUI elements from the ECNO framework

The implementation of the customized GUI can be found in the package `dk.dtu.imm.se.ecno.examples.workers.gui`, where the main class is `WorkersGUI`. We start explaining some of the simpler parts of that GUI, which directly use some of the GUI elements provided by the ECNO framework. In our example, this applies to the panel with the arrive and depart buttons for the selected worker at the buttom of the GUI (see Fig. 5.1 and Fig. 5.2). This panel is implemented by the class `ElementButtonPanel`, which is instantiated with some element of the ECNO application, and will have one button for each of the GUI event types in which the chosen element can be involved in.

The use of this `ElementButtonPanel` can be seen in the implementation of the `updateSelectedWorker()` method of the class `WorkersGUI`. This method is called, whenever the user selects another Worker in the GUI's "Select a worker" panel. Listing 5.1 shows the relevant part of the `updateSelectedWorker()` method; it creates the respective panel for the `selected` worker. In this context, **`this`** is the `WorkersGUI`, which extends the Swing `JFrame` class. As mentioned earlier, the class `ElementButtonPanel` comes from the ECNO Framework and extends the Swing `JPanel`; it is instantiated with the `engine` and the `selected` element – the worker in our example. In our example, the middle parameter is **`null`**; generally, this parameter can be used to provide an implementation `IPresentationUtil`, which would define custom labels for the panel itself,

[11] Note that this plugin project does not have a plugin.xml file! We deleted the plugin.xml that was generated by the EMF generator in this project since we use this example as a Java applications only – and not as a Eclipse application. The extensions from plugin.xml would conflict with the project from the introduction "dk.dtu.imm.se.ecno.example.workers" since it uses the same Ecore model with some extensions without using another namespace URI.

the buttons on it, and the tool-tips when the mouse is hoovering over the buttons. In our simple example, the default labels are fine; therefore, we do not provide an `IPresentationUtil` to the panel. As any Swing panel, the `ElementButtonPanel` can be added and removed from other Swing elements. But, we do not discuss the details here.

Once initialized, the `ElementButtonPanel` takes care of updating the possible interactions for the element and and the respective events, and the buttons associated with the respective event types update their enabledness fully automatically dependent on whether there is an interaction enabled for the element and the respective event type. We do not need to program anything for that.

In order to get an idea of what is going on behind the scenes and the interfaces of the ECNO framework making this mechanism work, we discuss some of the main concepts and classes underlying this mechanism anyway. The respective classes can be found in the ECNO core project `dk.dtu.imm.se.ecno.core` in the packages `dk.dtu.imm.se.ecno.gui` and `dk.dtu.imm.se.ecno.engine`. The actual buttons on the `ElementButtonPanel` are implemented by the class `ElementEventButton`, which are initialized with an engine, a presentation utility (optional), and an element and an event type. Then `ElementEventButton` will take care of tracking whether there are enabled interactions for the given element and event type, and update the enabledness of this button respectively. If the user presses the enabled button, the currently selected[12] interaction is executed. For properly updating the buttons when the possible interactions change, the `ElementEventButton` is derived from another class of the ECNO Framework, which is called `ElementEventController`; this class has all the infrastructure to register with the engine, and receive notifications when possible interactions change. To this end, the `ElementEventController` registers a so-called `IInvalidationListener` to the engine; but we do not explain the details here.

The class `ElementEventController` has also some methods that can be overridden by extending classes. For example, the method `getEvent()` can be overridden to add some pre-defined parameters to the event for which the interactions should be computed. This way, only those interactions are computed in which the event takes these parameters.

We will see how to use some of the above mechanisms in the following sub-section.

[12]Remember that, if there is more than one enabled interaction for the element and event type, the user can iterate through all the possible interactions, by using the SHIFT- and CTRL-key while pressing the mouse buttons.

5.5.2.3 Programming GUI elements

The main function of GUI elements is to properly update when the possible interactions change and to execute the resp. interaction when the user request it by some gesture (typically a mouse-click) at the GUI. The ECNO framework provides a mechanism which notifies some listeners, when an interaction becomes invalid or when the possible interactions that were computed by an interaction iterator do change. Note that this mechanism does not only issue notifications when interactions become invalid, but also issues a notification when new interactions become available.

The main interface is `IInvalidationListener`, which we explain in some more detail in this section. We do this by discussing the implementation of "Select a work item" panel (see Fig. 5.1 and Fig. 5.2). On the side, we explain how the ECNO Engine can compute interactions not only for a given event type, but it can compute interactions for a specific event with some preset parameters. In our case, the parameter is the name of the new job that should be created when the job is executed, where the name is passed as a parameter to the doJob event; remember that the ECNO model requires this name to be longer than 5 characters. Therefore, these interactions are possible only when the required workers are available and the user has entered a name to the text field that is more that 5 characters long.

Listings 5.2 to 5.5, show the most relevant parts of the class `Worklist Panel` which implement the "Select a work item" panel. We discuss these parts below.

Listing 5.2 shows the attributes that are used by class `WorklistPanel`. The attribute `engine` refers to the ECNO engine in which this GUI is running, and attribute `gui` refers to the `WorkersGUI`, which hosts the worklist panel. The attributes `textField`, `jobsComboBox`, and `button` represent the three widgets that are placed on the worklist panel, which are using the respective Swing classes. The attribute `buttonListener` is a class that will be associated with the `doJob` button in order to execute the selected job, when the button is pressed.

The most important attributes for our discussion here, are the last four attributes: The attribute `iterator` represents the interaction iterator, which keeps the possible interactions (representing executable jobs) for the worklist panel in the current situation. The attributes `doJob`, `jobParam`, and `nameParam` are referring to the ECNO meta model classes; they are initialized such that `doJob` points to the doJob event type of the ECNO model of the workers example, and `jobParam` and `nameParam` refer to the respective parameters job and name of that event type. We will discuss how these attributes are initialized shortly (see lines 7–19 in Listing 5.3).

Listing 5.3 shows an excerpt of the constructor of the class `Worklist Panel`. First, we discuss lines 7–19, which initializes the attributes for the doJob event type and its parameters. They are basically looked up from

Listing 5.2: Attributes of `WorklistPanel`

```
final ExecutionEngine engine;
final WorkersGUI gui;

private JTextField textField;
private JComboBox<InteractionItem> jobsComboBox;
private JButton button;

private ActionListener buttonListener;

private InteractionIterator iterator;

private EventType doJob;
private IFormalParameter jobParam;
private IFormalParameter nameParam;
```

the package with the ECNO model of the workers example; like in the EMF technology, an ECNO package is translated to a Java class, which represents this package and allows to look up and access its element types and event types. At first, all event types that are defined in the package are obtained and assigned to variable `events`. If there is an event type with name "doJob", this event type is assigned to the attribute `doJob`; likewise the parameters are looked up in this event type and assigned to the resp. attribute (note that we exploit the order in which these parameters are defined in the doJob event here to access the resp. parameter).

There are two other things, which need to be done in the constructor of the class `WorklistPanel`. First, we need to make sure that the possible interactions are recomputed whenever the user changes the text field that provides the name of the new job (leaving it to the model of the workers example whether the name would be legal or not). The actual update of the possible interaction is done in method `update()`, which is discussed below (see Listing 5.4). But, we need to make sure that this `update()` is called, whenever the user makes a change; to this end, we install an action listener to the text field, which calls the `update()` method in lines 26–30 of Listing 5.3. Another action listener (lines 40–43) makes sure that the current selected interaction is executed, when the `doJob` button is pressed. This action listener calls the `execute()` method which is discussed later (see Listing 5.5).

The most involved method of the `WorklistPanel` is the `update()` method. The implementation is shown in Listing 5.4. In this method, the current interaction iterator is maintained and registered with the ECNO engine so that the worklist panel is notified and properly updated when jobs become available or unavailable. The reason why this method is a bit more

Listing 5.3: Constructor of `WorklistPanel`

```
public WorklistPanel(ExecutionEngine engine,
    WorkersGUI gui, WorkersModel model) {
  super();
  this.engine = engine;
  this.gui = gui;

  doJob = null;
  List<EventType> events = model.getNamespace().getEventTypes();
  for (EventType type: events) {
    if ("doJob".equals(type.getName())) {
      doJob = type;
      break;
  } }
  if (doJob != null) {
    jobParam = doJob.getFormalParametersList().get(0);
    nameParam = doJob.getFormalParametersList().get(2);
  } else {
    throw new RuntimeException("DoJob event type not available");
  }

  ...

  textField = new JTextField();
  textField.setText("Some name");

  textField.addActionListener(
    new ActionListener() {
      public void actionPerformed(ActionEvent arg) {
        WorklistPanel.this.update();
    } });

  /* Add some other listeners for textfield update (mouse and focus) */
  this.add(textField);

  jobsComboBox = new WideComboBox<InteractionItem>();
  this.add(jobsComboBox);

  button= new JButton();
  button.setText("DoJob");
  buttonListener = new ActionListener() {
    public void actionPerformed(ActionEvent arg) {
      WorklistPanel.this.execute();
  } };
  button.addActionListener(buttonListener);
  this.add(button);
}
```

Listing 5.4: Update of `WorklistPanel`

```
void update () {
  if (enterUpdate()) {
    do {
      jobsComboBox.removeAllItems();
      Worker worklist = gui.getSelectedWorker();
      if (worklist != null) {
        if (iterator != null) iterator.unregisterListener(this);
        List<InteractionItem> interactions =
          new ArrayList<InteractionItem>();
        boolean success = false;
        do {
          Event event = engine.createInstance(doJob);
          if (event != null) {
            Parameter param = event.getParameter(nameParam);
            if (param != null) param.setValue(textField.getText());
            iterator = engine.getInteractions(worklist, event);
          }
          success = true;
          try {
            iterator.registerListener(this);
            while (iterator.hasNext()) {
              Interaction interaction = iterator.next();
              IChoice choice = interaction.getElementsChoice(worklist);
              event = choice.getEvent(doJob);
              Parameter jobP = event.getParameter(jobParam);
              if (jobP != null) {
                Object job = jobP.getValue();
                if (job instanceof Job) interactions.add(
                    new InteractionItem(interaction, (Job) job));
          } }
          } catch (InvalidStateException e) {
            iterator.unregisterListener(this);
            iterator.dispose();
            success = false;
          }
        } while (!success);
        for (InteractionItem item: interactions)
          jobsComboBox.addItem(item);
        if (!interactions.isEmpty()) {
          button.setEnabled(true);
          jobsComboBox.setSelectedIndex(0);
        } else {
          button.setEnabled(false);
      } }
    } while (!leaveUpdate());
} }
```

complicated is that this method will be called whenever possible interactions of the current interaction iterator change; and this can, in principle, happen concurrently to updating the possible interactions; and it can happen that while updating the interaction iterator "realized" that it has become invalid. The `update()` method needs to take care of these situations and a combination of them – using some of the infrastructure that the ECNO framework provides for this purpose.

Let us discuss some aspects of that implementation here. We start with the part taking care of maintaining and updating the possible interactions. First, in line 7 of Listing 5.4, the GUI unregisters as listener from the current interaction iterator, because the interaction iterator is about to be recomputed and updates from the outdated one are no longer relevant. Then (line 8), an empty list with new interactions items is created; these interaction items represent an interaction, but in such a way that these elements can be used as items of a `ComboBox`. Then (line 11–15), a new instance of a doJob event is created via the engine using the `doJob` attribute, and the name parameter of that event is set to the current value of the text input field of the panel. Then (line 16), an iterator for the current worker (variable `worklist`) and the event is created; and (line 20) the panel is added as a listener to this new interaction iterator.

Then (lines 21–30), the interaction iterator is used to fill the list of interaction items by iterating over the interaction iterator. Note that by concurrent changes, the interaction iterator might become invalid any time, in which case it would raise an exception when used. Therefore, this part of the computation is included in a try-catch block and a do-while loop, which takes care of this. If such an exception happens, the current interaction iterator is properly disposed of, and the current computation is marked as not successful – and the computation is started over again (line 36).

In the end (lines 37–43), the combo box is updated with the interaction items and the enabledness of the doJob button is set according to whether there is at least one job or not.

Note that all this computation is embedded into an if-statement with condition `enterUpdate()` (line 2) and a do-while loop (line 3–45) with condition `!leaveUpdate()`. This makes sure that the update is active only once at the same time; if another update is active already for this panel, the update finishes right away; in that case the condition `!leaveUpdate()` will make sure that the already running update is repeated in the end, since the current computation was overtaken by a later update. The `enterUpdate()` and `leaveUpdate()` are implemented using standard concurrent programming techniques in order to make them atomic; but, we do not discuss the details here.

At last, we discuss how to actually execute a job or the respective interaction, when the doJob button is pressed. This is implemented in the `execute()` method, which is shown in Listing 5.5. This code is quite sim-

Listing 5.5: Execute of `WorklistPanel`

```
void execute() {
  Object selected = jobsComboBox.getSelectedItem();
  if (selected instanceof InteractionItem) {
    Interaction interaction = ((InteractionItem) selected).interaction;
    if (interaction.isValid()) {
      try {
        interaction.execute();
      } catch (InvalidStateException e) { }
    }
  }
}
```

ple, basically calling the `execute()` method on the interaction (accessed from the currently selected `InteractionItem`). Again, due to concurrent behaviour the interaction might actually have become invalid when `execute()` is called; therefore, we explicitly check for its validity; but even then, it might happen that it becomes invalid right after that; therefore, we need to embed the call of the `execute()` method on an interaction into a try catch block. If the `execute()` returns without exception, we can be sure that the interaction was executed successfully; if the exception is raised, we can be sure that it was not executed at all (see Sect. 5.5.3).

Note that we do not need to initiate any update on the GUI in the `execute()` method. This will be taken care of indirectly by the ECNO engine. If some changes of the executed interaction change the possible interactions, the ECNO engine will notify the GUI, since the GUI had registered itself with the engine for notification.

5.5.2.4 Engine controllers

In our workers example, there are two panels which are *element controllers*, the "Select a work item" panel and the panel for "arrive" and "depart". An element controller is installed on one element, and typically has one or more listener installed, which notify it when the set of possible interactions changes.

In our case, these element controllers are installed explicitly on a worker when the worker is selected. In some ECNO applications, however, such element controllers should be installed automatically when elements are added to the engine or when the engine, becomes aware of new elements while computing interactions. For example, the generic ECNO GUI shows an ElementEventPanel for all the elements that are explicitly added to the engine. This is achieved by installing a *engine controllers* with an ECNO engine, which must implement the `IController` interface.

Listing 5.6: The `IController` interface

```
public interface IController {

  public void addElement(Object element);

  public void removeElement(Object element);

  public void elementEncountered(Object element);

  public void dispose();

}
```

This interface is shown in Listing 5.6. An implementation of these methods will allow the controller to install new element controllers when elements are explicitly added to the engine or when new elements are encountered while computing enabled interactions. It is completely up to these engine controllers what they do in these situations; but typically, they would install some element controllers on some of the elements – or remove them.

Any number of engine controllers may be installed with the ECNO engine. It is also possible that engine controllers remove themselves from the engine when they are no longer needed. When the engine terminates, and the engine controller is still installed, the engine will call its `dispose()` method. Actually, the ECNO engine terminates[13] when the last controller is removed from the engine.

Typically, the GUIs for an ECNO application will register itself as an engine controller. Our `WorkersGUI` as well as the default `ECNOGUI`, both register themselves as engine controllers (in their constructor). But, it is possible that GUIs are not engine controllers, and there might be engine controllers that are not GUIs. One example would be an engine controller that installs an element controller that automatically issue the execution of some interactions when they become enabled.

The engine controllers can be either installed programmatically or via a configuration, which depends on whether the ECNO application is run as a Java application or an Eclipse application. This is discussed in the following section. It is possible to install or remove new engine controllers to a running engine at any time. In our workers example, the user pressing the "New worklist panel" will create a new Worklist GUI, which then is added

[13]Note that when an ECNO application is running as Eclipse application, the ECNO engine view will always be installed as an engine controller; therefore, the ECNO applications running as Eclipse applications terminate only upon an explicit request from the application itself or when the user presses the close button for an application in the ECNO engine view.

Listing 5.7: The main method of class `SettingWorkersGUI`

```
public static void main(String[] args) {
  ExecutionEngine engine = ExecutionEngine.createNewInstance();
  WorkersModel model = WorkersModel.getModel(engine);
  engine.addPackageAdapter(model);
  if (!engine.resolveNamespaceImports()) {
    System.err.println("Package imports could not be resolved");
  }
  new WorkersGUI(engine, model);
  Setting instance = new Setting();
  // Note: to create an instance from here, I made the method
  //       createInstance in the automatically generated class
  //       Setting visible for the package.
  instance.createInstance(engine);
}
```

to the ECNO engine; pressing the "Close window" icon of that window, will terminate the Worklist GUI and remove it as a engine controller from the ECNO engine.

5.5.2.5 Configuring ECNO applications

When an ECNO application is started, some engine controllers must be initially installed with the ECNO engine. We call the initialisation of the controllers of an ECNO application the *configuration* of the ECNO application. By default, an ECNO application will start up the standard ECNO GUI, which is an ECNO engine controller. In many cases, however, we need to install some specific engine controllers for an application. In particular, this applies when an ECNO application should run with a custom-made GUI.

In this section, we discuss how to configure an ECNO application. Actually the way of configuring an ECNO application depends on whether we run the ECNO application as a Java application or whether we run it as an Eclipse application.

Configuring Java applications We start explaining how to configure an ECNO Java application. In this case, the class generated for the instance needs to be modified manually[14], adding some code that starts up and installs the initial engine controllers programmatically.

[14]In the future, we might introduce a way to do that in a more elegant way. But, since running ECNO as a Java application will probably be the exception, this does not have high priority.

In order to explain how to do this, let us have a look at our workers example again. Listing 5.7 shows an excerpt of the class `SettingWorkersGUI` in the package `dk.dtu.imm.se.ecno.examples.workers.instances` of the project `dk.dtu.imm.se.ecno.examples.workers.worklistgui`. The class `SettingWorkersGUI` is manually written, but it is similar to the automatically generated class `Setting` from the same package. Listing 5.7 shows the `main()` method, which is very similar to the one in the automatically generated class `Setting`; the only difference is in line 8, where the `WorkersGUI` is started up instead of the standard `ECNOGUI`. Note that the constructor of the class `WorkersGUI` takes care of registering itself with the engine; therefore, there is no code in the `main()` method that would register the `WorkersGUI` with the engine. Note also that we made the private method `createInstance()` of the automatically generated class `Setting` accessible for the package, so that we could use this method here for actually creating the setting.

On the side, you can see in line 2 of Listing 5.7 how to obtain an ECNO model and how to register it with the engine by calling the static `getModel()` method of the class `WorkerModel`, which represents the ECNO model for workers. Note that this methods is called with an instance of an engine as a parameter, which makes it possible to use the same model with different ECNO engines. The relevant ECNO models need to be registered with the ECNO engine before the actual instance of the model is loaded or created in the engine.

Configuring an Eclipse applications For ECNO Eclipse applications the *configuration* is done in two steps. First, the ECNO application needs to implement some *controller configurator* and to register it as an extensions to Eclipse. Then, the actual configuration is defined in a *start configuration* as discussed in Sect. 5.4.1 already – referring to the controller configurator that was defined in the first step. Here, we discuss how to implement a *controller configurator*. Since our workers example is an ECNO Java application only, we need to use another example here: the Petri net example, in which the GUI is set up to work with the graphical editor of the Petri net so that the token game is shown as tokens in the graphical editor, which we had used in Chapter 2; the use of the example was explained in Sect. 2.1.2.

Here, we use this example to discuss how to implement a controller configurator, how to plug it in to Eclipse, and how to create a start configuration. The code for this example can be found in the project `APetriNet EditorIn15Minutes.ecno.gui` in the only package `dk.dtu.imm.se.ecno.example.petrinets.gui`.

Listing 5.8 shows the main part of the implementation of the `Petrinet GUIConfigurator`. In order to be used as a controller configurator later, this class must implement the interface `IControllerConfigurator`. The

Listing 5.8: Implementation of the `PetrinetGUIConfigurator`

```
public void initializeControllers(ExecutionEngine engine,
    ControllerState state) {
  this.engine = engine;
  if (state instanceof ObjectReference) {
    EObject object = ((ObjectReference) state).getObject();
    if (object instanceof Diagram) {
      diagram = (Diagram) object;
      EObject element = diagram.getElement();
      if (element instanceof PetriNet) {
        petrinet = (PetriNet) element;
        listener = new PetrinetContentListener(engine);
        petrinet.eAdapters().add(listener);
      }
      ResourceSet resourceSet = object.eResource().getResourceSet();
      IOperationHistory operationHistory = OperationHistoryFactory.
        getOperationHistory();
      domain = GMFEditingDomainFactory.getInstance().
        createEditingDomain(resourceSet, operationHistory);
      domain.setID("APetriNetEditorIn15Minutes.diagram.EditingDomain");
      try {
        editor = PetriNetDiagramEditorUtil.
          openDiagram(diagram, operationHistory);
      } catch (PartInitException e) {
        e.printStackTrace();
  } } }
  new ECNOGUI(engine);
  engine.addController(this);
}
```

interface `IControllerConfigurator` has one method `initializeControllers()`, which takes two parameters: the engine which it is supposed to configure, and some state object (which is provided by the start configuration and is discuss later). In the implementation of the `PetrinetGUIConfigurator`, this state object is assumed to refer to the Petri net diagram corrsponding to the underlying Petri net of the simulation, so that the GUI can be started with this diagram to graphically visualize the token game.

Basically, programming the set up of the configuration follows the same structure as we have seen for Java applications. Let us briefly discuss the method `initializeControllers()` shown in Listing 5.8. First, the `engine` is stored for future reference, since this controller configurator will later also serve as an engine controller. Then, the state object is extracted, checked whether it is an `ObjectReference` which is a type from ECNO, which allows the state to refer to any other EMF object in any resource.

Then, it is checked, whether this object is a `Diagram` object, which is a class from the GMF framework representing diagrams. For this diagram, the underlying model `element` is extracted and it is checked whether it is a Petri net. If it is, a `PetrinetContentsListener` is installed, which takes care of adding and removing transitions that are added to or removed from the diagram also to the ECNO engine, so that the ECNO engine and the controllers registered with it become aware of these added or removed elements. The implementation of this class is straight forward, and we do not discuss this here.

The most important part is starting up the GMF editor with the diagram. In order to integrate the command stack resp. operation history of the GMF editor with the command stack of the ECNO engine, we need to properly set up the domain with a specifically created operation history and then open the diagram, with the `PetriNetDiagramEditorUtil`, which is a utility class that was generated by GMF. Note that we also needed to make some manual changes to the automatically generated GMF editor, since normally the editor can be opened with an input file only. Here, we start it directly with a diagram object. But, we do not discuss these details here, since this is very specific for the GMF technology, which is not our focus here.

In the end, the default ECNO GUI is instantiated and installed as an engine controller, and the controller configurator installs itself an engine controller – implementing the interface *IController*. Actually, the `PetrinetGUIConfigurator` does not do much as an engine controller; the only implemented method is `dispose()`, which properly takes down the graphical editor when the ECNO engine is shutting down and removes the listener from the Petri net. But, we do not discuss the details here either.

In essence, the method `initializeControllers()` is used to install some engine controllers on the engine and can start some other parts of the software, such as the graphical Petri net editor in our case. For properly doing this, the configurator can also use some additional information, which is provided in the start configuration; in our example, this is the diagram for which the graphical editor should be started.

As mentioned before, it depends on the start configuration which controller configurator is used when the ECNO application is started. The configuration will do this by referring to a unique URI under which the controller configurator is registered with the Eclipse. To this end, we need to plug in our `PetrinetGUIConfigurator` to Eclipse, for which ECNO provides a specific extension point. As usual in Eclipse, declaring such an extension is done in the project's "plugin.xml" file. Listing 5.9 shows the part of the `plugin.xml` file in which the class `PetrinetGUIConfigurator` is plugged in as a controller configurator extension with the URI `APetriNet EditorIn15Minutes.Simulator.GUI`. The **`class`** attribute refers to the class `PetrinetGUIConfigurator` with its fully qualified name, and the `uri`

Listing 5.9: Plugging in `PetrinetGUIConfigurator` to Eclipse

```
<extension
    point="dk.dtu.imm.se.ecno.eclipse.engine_controller_configurator">
  <configurator
    class=
      "dk.dtu.imm.se.ecno.example.petrinets.gui.PetrinetGUIConfigurator"
    uri="APetriNetEditorIn15Minutes.Simulator.GUI">
  </configurator>
</extension>
```

Listing 5.10: Start configuration `a_net.behaviourstates`

```
<behaviourstates:BehaviourStates
    xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:behaviourstates=
      "http://ecno.se.imm.dtu.dk/ecno/save/bahaviourstates">
  <states>
    <state xsi:type="behaviourstates:DefaultContainer"/>
    <element href="a_net.petrinets#/"/>
  </states>
  <controllerConfig uri="APetriNetEditorIn15Minutes.Simulator.GUI">
    <state xsi:type="behaviourstates:ObjectReference">
      <object href="a_net.petrinets_diagram#_5khEse7qEeK8p9ZDqdJC5A"/>
    </state>
  </controllerConfig>
  <packages>PetriNets.ECNO</packages>
</behaviourstates:BehaviourStates>
```

is the unique URI, we choose for this plugged in controller configurator.

At last, let us have a brief look at how to refer to such a controller configurator in a start configuration. In Sect. 5.4.1, we have seen already how to start an ECNO Eclipse application from such an start configuration. You find examples in the project `APetriNetEditorIn15Minutes.runtime` in folder "run". For example, there is "a_net.behaviourstates". We can start the application as described earlier in Sect. 5.4.2, by right-clicking on this start configuration and selecting "ECNO → Start ECNO Engine".

Here, we have a brief look into this start configuration. The start configuration can be opened and edited in the EMF tree editor. For discussing the contents of the start configuration, however, it is easier to have a look at its XML representation, which is shown in Listing 5.10. The first element refers to the actual Petri net and says that all the elements contained in it

will start in the initial state of the element types behaviour (therefore it is called a default container). The Petri net that the states element refers to is the top-level element in the file "a_net.petrinets" – the Petri net contained in the file.

After the states, the controller configurator that should be used for setting up the initial configuration is defined. This refers to the unique URI `APetriNetEditorIn15Minutes.Simulator.GUI` of the controller configurator that we have implemented and plugged in above. And the state object passed to the configurator is the diagram element of the file "a_net.petrinets_diagram". Note that there is a last element in this start configuration, which is called "packages". This tells the ECNO engine which ECNO packages it should load – here, the ECNO package for PetriNets is required. Note that the ECNO code generation will plugin an ECNO package to Eclipse when the code is generated fully automatically, provided that the gen model has the attribute "PackageAdapter Factory Class Name" set in the ECNO gen model. Therefore, we can refer to the packages URI without explicitly declaring this extension – the ECNO code generator takes care of that.

If you have a look into the source code of the project `APetriNetEditorIn15Minutes.ecno.gui`, which we used in this example, you will find two additional classes: class `FireTransitionController` and class `PetrinetGUIConfiguratorWithAutoFire`. These are not relevant for our discussion here, but we will use them later for discussing how to automatically fire enabled transitions – or more generally, how to automatically execute enabled interactions, which will be discussed in Sect. 5.5.3

5.5.2.6 Future plans: A DSL for Customized GUI

In the sections above, we have discussed the mechanisms that can be used for programming specific GUIs for specific ECNO applications; in particular, we have discussed engine controllers, element controllers, and interaction listeners as the core mechanisms that can be used for that purpose. Programming such GUIs is still a bit technical, but should not be difficult at all when following the basic principles discussed in this section.

Anyway, the need of programming a GUI seems to be an anachronism, in particular in the context of the ECNO approach with the focus on modelling software. Therefore, we plan to implement a notation that will allow us to model GUIs for ECNO applications from which the program code for the GUI can then be generated fully automatically. But, this is a long-term plan since this is not the main concern of the research on ECNO.

5.5.3 Transactions and automatic controllers

As mentioned earlier, the ECNO engine makes sure that interactions are computed and executed in a thread-safe way, and controllers computing interactions are automatically notified, when possible interactions change. And, as long as all behaviour is coming from the ECNO models, the interactions of an ECNO application are executed in a transactional way (following the ACID principle as discussed in Sect. 5.5.1). If changes are made from other parts of the software, some extra measures need to be taken in order to make sure that transactionality is not compromised.

In this section, we briefly discuss how other parts of the software can make changes in such a way that transactionality of the interactions and the other changes are maintained. Moreover, we discuss how to implement element controllers that automatically execute an interaction on an element, when it becomes enabled.

5.5.3.1 Transaction management

An ECNO application will smoothly work together with other software which makes its changes on the objects underlying the ECNO application directly – without using the ECNO framework. All ECNO controllers will be properly updated when possible interactions change[15] due to such changes. In order to obtain transactionality, however, changes made outside an interaction need to be explicitly made transactions.

To this end, the ECNO engine is run with a transaction manager; the type of the transaction manager depends on the configuration of the ECNO engine. The transaction manager, can be obtained from the running ECNO engine and can then be used for starting and finishing a transaction. And the transaction can, dependent on the type of the transaction manager, be used to lock all the elements that are used and changed within the cope of the transaction. Up to now, the ECNO engine runs with a default transaction manager which uses a very simple locking mechanism.

In the following, we discuss the main functions for obtaining and using the transaction manager of the ECNO engine:

- Form an ECNO `engine`, its `TransactionManager` can be obtained by calling `engine.getTransactionManager()`.
- A `transactionManager` obtained this way can then be used to create a new transaction by `transaction = transactionManager.startTransaction();`
- It depends on the configuration of the engine, which kind of transaction is returned. In the standard configuration, the `transaction` is of

[15] Actually, this depends a bit on the on the underlying OO-technology. But for EMF, the updates are fully supported.

kind `LockingTransaction`. In that case, by casting the transaction to `LockingTransaction` can be used to lock some objects by calling `transaction.lock(objects)`, where objects is the `Set` of objects to be locked. For a `LockingTransaction`, all objects that are involved (read and write) during the transaction must be locked in a single go before the actual changes are made.

- After the transaction is started and the involved objects are locked, the programme can make any change on the locked objects. Once all changes are made, the transaction should be terminated by calling `transactionManager.stopTransaction(transaction)`.

Note that the transaction manager might raise some exceptions, when the engine is terminated while a transaction is still running. And the lock operation might cause the current thread to wait until the locks for all the required objects could be acquired. The default locking transactions will raise an exception if the `lock()` operation is called twice for the same transaction. The locking mechanism will, however, never block the system forever, if the application makes sure that all transactions that have acquired a lock will eventually be stopped as discussed above. Generally, it is recommended that transactions should run fast, once the locks are acquired.

Up to now, the ECNO framework provides this simple transaction manager only. This is mostly used as a proof of concept and for completeness sake. In the future, the ECNO framework might be equipped with some more advanced and flexible transaction managers – driven by concrete needs for some applications.

Note that it is recommended to use transactions for all kinds of changes that are made outside interactions, even when the application itself does not require transactional execution. The reason for this recommendation are efficiency considerations: any individual change of some element for which some controllers are installed might cause the re-computation of all possible interactions in which this element might be involved; this causes a lot of computations for every single change of an object; if these changes are made under the control of a transaction, the possible interactions are re-computed only once, viz. when the transaction is finished after all the changes have been made. This can reduce the number of necessary re-computations of possible interactions significantly and, this way, performance is increased.

5.5.3.2 Automatic execution of interactions

Next, we briefly show how to implement an element controller that automatically executes interactions once they become enabled. To this end, we continue our Petri net example and add an element controller that automatically fires an enabled transition with some random delay (with nega-

tive exponential distribution[16]). The code for this example can be found in the package `dk.dtu.imm.se.ecno.examples.workers.instances` of the project `dk.dtu.imm.se.ecno.examples.workers.worklistgui`. Here we discuss the element controller `FireTransitionController`, which is responsible for computing the enabled interactions and for executing them. This element controller is automatically installed for each transition by the engine controller `PetrinetGUIConfiguratorWithAutoFire`, which also is a controller configurator. But we do not discuss the class `PetrinetGUI ConfiguratorWithAutoFire` here, since we have discussed engine controllers and controller configurators in Sect. 5.5.2 already.

Listing 5.11 shows the main parts of the `FireTransitionController`, which is the element controller for transitions and responsible for computing and initiating the execution of possible interactions for fire events on transitions. This controller is instantiated with the execution engine, a transition and the event type as a parameter – in our example the event type will be the fire event. Most of the behaviour of this element controller is inherited from the `ElementEventController` from the ECNO framework. The only change is overriding the `setEnabled()` method. If this method is called with the `enabled` parameter set to true, the corresponding `interaction` is obtained – by an attribute from the super class `ElementEventController`; if the interaction is not **`null`**, a thread is created in which this interaction will eventually be executed. This thread is called `FireTransitionThread` and is discussed below; it takes three parameters: the engine, the interaction, and the firing rate (number of executions per second, here we choose rate 1.0/s). Note that it is very important not to call `interaction.execute()` directly in the `setEnabled()` method of the element controller since this would result in an infinite loop of updating interactions and executing them. So, even if the interaction is supposed to be executed immediately, this execution must be delegated to another thread – either a newly created one or some worker thread[17]. After creating the new thread, the thread is started right away.

Listing 5.12 shows the implementation of the class `FireTransition Thread`. The constructor is used only for storing the thread's parameters, the engine, the interaction, as well as the execution rate. The main part of

[16]We use (negative) exponential distribution since it is memoryless. Though this does not matter too much for our toy simulator, being memoryless has a theoretically appealing property: The ECNO framework sometimes invalidates interactions that actually are still valid; in that case, a completely identical interaction is computed again, and our controller will start its execution with a random delay again. Since the distribution for the execution delay is memoryless, it does not make any difference that we recompute an identical interaction and start it with a random delay again.

[17]Note that we chose to create a new thread for each interaction in this implementation. This is not very inefficient, but it makes the code simpler and easier to explain. In realistic applications, you would probably delegate this to a fixed set of worker threads. But, since this is standard thread programming, we do not discuss this in more detail here.

Listing 5.11: Element controller `FireTransitionController`

```
public class FireTransitionController extends ElementEventController {

  public FireTransitionController(ExecutionEngine engine,
      Transition transition,
      IEventType eventType) {
    super(engine, transition, eventType);
  }

  @Override
  protected void setEnabled(boolean enabled) {
    if (enabled) {
      Interaction interaction = this.interaction;
      if (interaction != null) {
        // NOTE: By now means call interaction.execute() here!!
        // We start a thread in which the execution is done instead:
        Thread thread =
          new FireTransitionThread(engine, interaction, 1.0);
        thread.start();
  } } }

}
```

the implementation is in the `run()` method. First of all, the random delay in milliseconds is computed using the (negative) exponential distribution for the given rate. Then, the thread sleeps for the computed delay. Note that due to concurrent user operations or due to other transitions firing automatically in the meantime, it might be that the interaction associated with the thread is no longer valid, when the thread wakes up again; it might even happen that the ECNO engine is no longer running at that time. Therefore, we first check for that. Only if the engine is still running and the interaction is still valid, the thread issues the execution of the interaction. Note that even though enabledness of the interaction was checked already, due to concurrent threads, the current interaction might be invalidated before it actually is executed – or the engine might be stopped before the execution – in which case calling the execute method will raise an exception. Therefore, we need to catch the respective exception. Independently of whether the interaction was executed or not, the thread terminates.

Note that we do not need to take any other measures concerning multi-threaded execution since the execution of interactions is implemented in a thread-safe way in the ECNO engine – as discussed in Sect. 5.5.1.

Altogether, automatically executing enabled interactions is quite easy: in the `setEnabled()` method of the respective element controller, schedule the execution of the computed interaction. The most important issue is that the

Listing 5.12: Firing thread `FireTransitionThread`

```
private class FireTransitionThread extends Thread{

  private ExecutionEngine engine;
  private Interaction interaction;
  private double rate;

  FireTransitionThread(ExecutionEngine engine,
      Interaction interaction,
      double rate) {
    this.engine = engine;
    this.interaction = interaction;
    this.rate = rate;
  }

  public void run() {
    double random = Math.random();
    long delay = (long) (-1000 * ( Math.log(1-random) / rate));
    try {
      sleep(delay);
    } catch (InterruptedException e) {}

    if (!engine.isExiting() && interaction.isValid()) {
      try {
        interaction.execute();
      } catch (InvalidStateException e) {}
} } }
```

interaction must not be directly executed in the element controller by which it was computed – this would result in an infinite recursion. The execution of the interaction must be delegated to another independent thread. Of course, it is not necessary to create a new thread for each possible interaction – we created a new thread for each new interaction for simplicity only.

5.5.4 Advanced features

In the sections above, we have discussed the main features of the ECNO Programming Framework only. Actually, there are many more features and some pipes and whistles that could be used for tuning and improving ECNO applications. In order not to loose focus, we do not go into more technical details. Instead, we briefly name two important features of ECNO, which are actually at its core. But, we do not go into details here. This overview should provide a starting point for further investigations only.

5.5.4.1 Package adapters

From an ECNO package with its coordination diagrams and ECNO nets for the different elements, the ECNO code generator, generates a so-called package adapter, which is then used by the ECNO engine for executing the defined behaviour. These package adapters make it possible to use ECNO with virtually any object-oriented technology – the ECNO engine just needs an adapter to look up the information for the coordination and the local behaviours. We did not discuss any details of these package adapters yet, because they work behind the scenes as long as we stick to EMF-models as the underlying object-oriented technology.

Actually, the package adapter is defined as an interface `IPackageAdapter` in the core project of ECNO `dk.dtu.imm.se.ecno.core`, which is completely independent of EMF and even independent of Eclipse. It basically has methods that, for a given object, allow to determine its ECNO element type, to create an object that represents the element's local behaviour when the element is encountered by the ECNO engine for the first time, and a method to compute the set of objects an element is linked to with respect to a reference in a given situation. Moreover, it allows to create event instances for the event types of that package – with the respective values for the event's parameters.

In addition, the package adapter gives access to the element and event types that are defined in the respective ECNO package. This representation, basically, follows the meta model of ECNO, which is discussed in Sect. 5.6.

Typically, these package adapters would be generated automatically for some specific underlying object-oriented technology. By default, ECNO supports the EMF technology. But, such a package adapter can also be programmed manually – we used that in the very early stages during the

development of the ECNO engine.

5.5.4.2 Programming local behaviour

As discussed above, the package adapter must be able to create an object that represents the local behaviour of an element. This object must implement the interface `IElementBehaviour` from the ECNO core. Basically, this behaviour object says in any given situation, which `IChoice` would be possible for that element, where the choice defines which events are involved and which changes would be made when the choice is taken. Due to behaviour inheritance, however, the details are a bit more involved – a choice of an element would be a a list of choices for each element type on the element's type hierarchy.

In practice, we would probably not want to program the local behaviour for an element type manually. But, this interface allows us to use other notations than ECNO nets for modelling local behaviour, if we want to – for people who are not so fond of Petri nets, for example.

5.5.4.3 Wrapping the execution of interactions

In the context of some existing applications, it is sometimes necessary, to wrap the execution of an interaction in order to do some preparations or maintenance work before and after the execution of the interaction by the engine. To this end, the engine allows to register factories that wrap the execution of an interaction into an `IInteractionExecutionCommand`. It is even possible to wrap the execution of the interaction several times for different purposes.

Actually, the ECNO Framework uses these wrappers for its own purposes already. For example, ECNO's transaction mechanism is implemented this way; and also the undo and redo mechanism for ECNO Eclipse applications is realized by using wrappers – the execution of an interaction is wrapped into a so-called recording command.

This way, it is possible to use the ECNO engine in a completely different transactional framework or to add some new functionality to the execution of interactions. Using these wrapper factories, would of course require some more detailed understanding of the wrapping mechanisms and the order in which ECNO applies its wrappers for achieving transactionality.

Up to now, the ECNO engine implements a basic and very simple framework for wrapping the execution of interactions, the design of which was driven by the needs of the ECNO framework itself. For some more sophisticated applications some additional functionality for the wrapping of commands might be needed. But, this will be added to the ECNO engine when the resp. needs arise.

5.6 ECNO meta model

At last, we make the concepts of ECNO's coordination diagrams a bit more explicit by presenting and briefly discussing the meta model of coordination diagrams. This meta model is defined by a set of Java interface in the package `dk.dtu.imm.se.ecno.core` of the ECNO project `dk.dtu.imm.se.ecno.core`.

Here, we actually do not discuss these interfaces, but we discuss an implementation of these interfaces, which uses EMF as its underlying object-oriented technology. This has several reasons: First and foremost, we can represent this meta model as an Ecore diagram; second, the structure of the EMF implementation is only a mild variation from the actual meta model as defined by the Java interfaces; third, the EMF implementation of coordination diagrams serves as a reference implementation of coordination diagrams; and lastly, the Ecore model for coordination diagram explicitly establishes the relation of the ECNO concepts to the concepts of the underlying object-oriented technology, the Ecore meta model itself.

Figure 5.3 shows the meta model of ECNO coordination diagrams in magenta, and relates it to the concepts from the Ecore model (marked with "from ecore" and shown in yellow). This meta model can be found in the folder `model` in the ECNO project `dk.dtu.imm.se.ecno.model`.

The main concepts are the `ElementType`, which contains `CoordinationSets`, which in turn are associated with a set of `Synchronisations`. The `Synchronisation` itself, however, is a part of the `Reference`, which is associated with the `ElementType`.

The `ElementTypes` are contained in a `Package`; the class `Package` is, actually, a minor deviation from the core model, where this class is called `Namespace`. The `Package` (or `Namespace`) contains also the `EventTypes` and `EventTypeExtensions` with their respective `FormalParameters`.

`ElementTypes`, `EventTypes` and `EventTypeExtensions` can be derived from other types, which is reflected by the feature **`super`**. As you might remember, ECNO allows multiple inheritance on `EventTypeExtensions` and each extended type is referred to by a name; therefore, the feature **`super`** is actually a qualified association. Since qualified associations are not an explicit concept of Ecore, this is represented by a standard pattern for modelling qualified associations in Ecore by using Mappings, which we do not discuss in detail here. Basically, the qualified association from the `EventTypeExtension` to itself is represented by the reference with name super to class `String2EventTypeExtensionMap`, where the attribute `key` is the qualifying attribute; the reference value points to the target of the qualified association.

Note that there is also a concept of `ImportedType` in this model. This is specific to the EMF implementation of coordination diagrams. This allows referring to types of other ECNO packages even when they use different

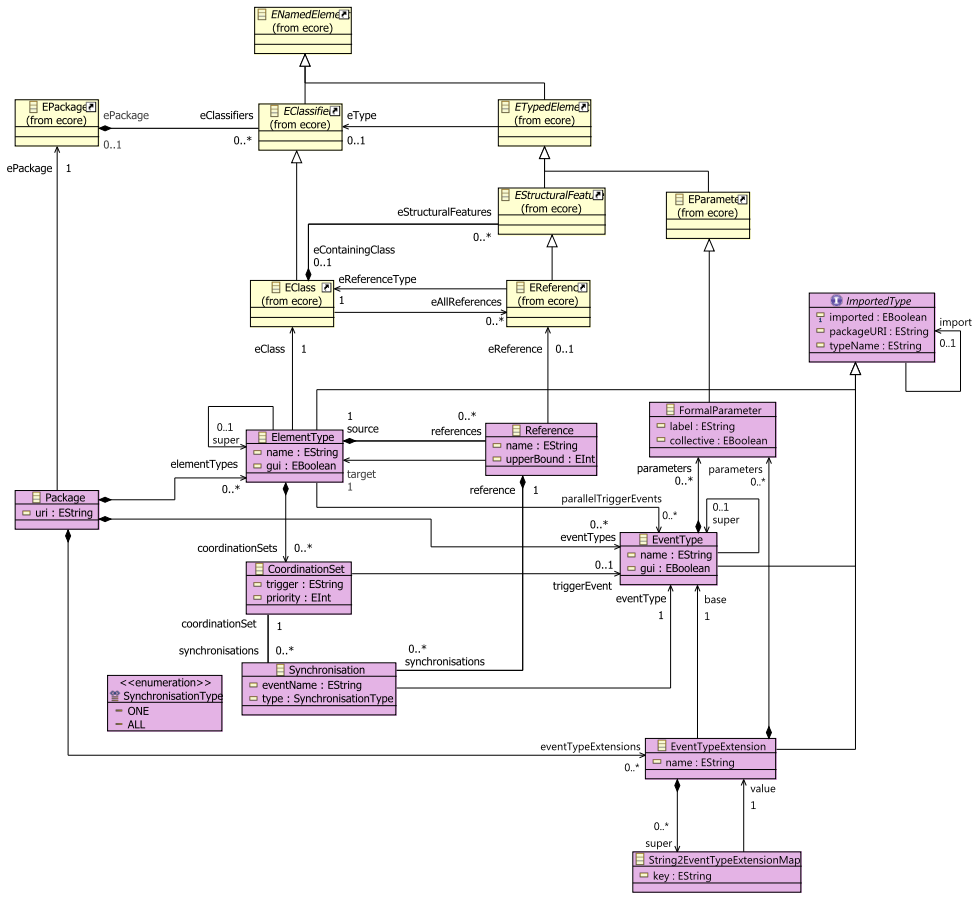

Figure 5.3: Meta model for coordination diagrams (EMF implementation)

underlying object-oriented technologies. The way the import mechanism is realized is left to the implementation, however. Therefore, we do not discuss this further here. In case of an import from a package from the EMF implementation, an import can be realized simply by referring to the resp. type in the other package (in which case the name of the type and the package URI can automatically be derived). In case of other technologies, the package URI and the name of the type must be set manually.

Note that, in the EMF implementation of coordination diagrams, `EventTypes` and `EventTypeExtensions` appear not to have anything in common. In the underlying core model, however, both are derived from so-called `EventKinds`.

Chapter 6

More examples

In this chapter, we discuss two additional examples of ECNO applications in order to provide some more insights into the characteristics and the modelling power of ECNO.

The first example is an ECNO model of so-called signal-event nets [50, 15], which are an extended form of P/T-systems. The main purpose of this example is to show that the definition of the semantics of S/E-nets in terms of ECNO is a straight forward extension of the ECNO semantics for P/T-systems. The discussion of the ECNO model is based on an article in the Peri Net Newsletter [34].

The second example is an ECNO model of a workflow engine. The code generated from these models – together with a manually implemented GUI for a worklist – allows enacting workflow models, which consist of models concerning different aspects of the workflow. This example should demonstrated that ECNO allows modelling larger scale software and to automatically generate code from it. This workflow engine is the result of a master's project of Jesper Jepsen [26]. The main purpose of this project was a first evaluation of ECNO with a larger example; but the workflow engine might be considered as a contribution in its own right. Actually, this example also closes the circle: the modelling concepts of ECNO were very much inspired by some ad hoc modelling concepts and ideas that we had used in AMFIBIA [1, 2] for capturing the essence of business process modelling concepts (see discussion in as discussed in Sect. 1.4); in the ECNO workflow engine, we used ECNO to rephrase the ideas of AMFIBIA again.

By contrast to the examples that we had discussed in the earlier chapters, the discussion in this chapter will not go into all modelling details anymore.

6.1 An ECNO semantics for signal-event nets

Signal-event nets (*SE-nets*) are an extended form of Place/Transition systems (P/T-Systems), which we discussed in Sect. 2.1 already. The major

syntactical difference between SE-nets and P/T-systems is that, in SE-nets, there may be arcs between transitions, which are called *signal arcs* [50, 15].

The meaning of these signal arcs is the following: If there is a signal arc from some transition t_1 to transition t_2 and transition t_1 fires and t_2 can fire together with t_1, then t_2 is supposed to fire synchronously together with t_2; if t_2, however, is not enabled together with t_2 in the current marking, t_1 can fire without t_2. It would, however, be okay if t_2 fired on its own (i. e. without t_1 firing) provided that there is no signal arc from t_2 to t_1. Of course, there are some subtle issues when both transitions, t_1 and t_2, can fire in a given marking, but not both can fire together. Things get even more involved in case of chains of signal arcs, and even more involved when there are cycles of signal arcs (see [50, 15]).

Fortunately, we do not need to dive into this here. It is enough for us to formulate the rule above that t_2 needs to fire together with t_1, if t_2 is enabled together with t_1. All the subtleties will be taken care of by the semantics for P/T-systems that we had defined earlier already (see Sect. 2.1.3.2 and Fig. 2.4 on page 33). Therefore, we discuss only those parts of the coordination diagram that change – the local behaviour for the elements does not need to be changed at all. Figure 6.1 shows the part of the coordination diagram that needs to be changed in order to cover the semantics of SE-nets. The two references from the Arc to the right indicate that these references to Place and their coordination annotations do not change as compared to the diagram from Fig. 2.4. Only the Transition and Arc are affected. These changes are discussed below.

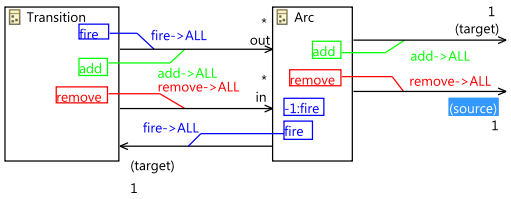

Figure 6.1: Coordination diagram: Global behaviour of SE-nets

The most important change is that, now, the event fire is propagated to all the out-going Arcs, and the Arc has coordination sets for the fire event now. Actually, there are two coordination sets, one with standard *priority* (which is 0) and one with priority -1. The coordination set with standard priority requires that all transitions[1] to which the arc points to participate

[1] Note that the the associated coordination annotation is quantified with ALL; since each arc always has one target, this coordination annotation requires that "the" transition the arc points to participates in the fire event.

in the fire event too. The two coordination annotations concerning event fire between the Transition and Arc together, actually reflect the rule that transition t_2 should fire together with t_1, if t_2 is enabled and when t_1 fires. The coordination set with priority -1 takes care of the situation that t_2 could not fire together with t_1. Since this coordination set has no coordination annotations attached, the fire even is not propagated at all. But, this coordination needs to have lower priority, since otherwise an enabled t_2 would not be forced to fire together with t_1.

Altogether, this shows that a minor twist (see the minor difference between Fig. 2.4 and Fig. 6.1) can take care of capturing the semantics of SE-nets as well. Moreover, this is our first example where priority of coordination nets come into play – without giving the empty coordination set for event fire lower priority, we would not enforce an enabled t_2 to fire together with t_1. They could fire together or not – the choice would be nondeterministic (erratic), which would not correspond to the actual semantics of SE-nets.

6.2 Workflow engine

All examples that we discussed in this report up to now, are relatively small and simple. They are well-suited to discuss the principles of ECNO and to convey the essence of ECNO and its use. In order to evaluate ECNO in a more realistic setting, we need to consider more realistic systems. To this end, Jesper Jepsen set out to develop a workflow engine based on ECNO in his master's project [26]. We discuss the main idea of the ECNO models underlying this workflow engine in this section.

6.2.1 AMFIBIA: A recapitulation

The workflow engine developed by Jesper Jepsen is based on the ideas and concepts of AMFIBIA [1, 2], which – as mentioned in Sect. 1.4 already – in turn has strongly inspired the development and concepts of ECNO. Therefore, we briefly recapitulate the concepts of AMFIBIA in this section first.

AMFIBIA is an acronym for *A Meta-Model for Integrating Business Process Modelling Aspects*. AMFIBIA set out to identify the main concepts of business process models independently from any specific modelling notation, and without a focus on or bias towards any specific aspect of business process modelling, such as *control*, *information* or *organisation*. Actually, AMFIBIA is open for adding new aspects, if need should be. AMFIBIA captures the concepts of business process models and their relation by a bunch of class diagram. This can, basically, be considered as an ontology of the concepts of business process models. But, AMFIBIA went one step further

in that it not only provides terminology, but also defines the behaviour of the concepts. To this end, AMFIBIA followed the following steps:

- AMFIBIA does not only define the concepts of business process models; it also defines the concepts for their execution, i. e. it defines an instance or a runtime model.
- AMFIBIA makes explicit which kind of events in general take place when executing a process, such as initiating a new instance of a process (which we call case), starting a task of a process in a case (which we call activity), finishing an activity, finishing a case, and some more.
- AMFIBIA defines when which kind of event is allowed to happen, and who needs to be involved in these events.

If this sounds familiar, this is not a coincidence: ECNO is made for modelling these kind of things. At the time, AMFIBIA had used its own ad hoc notation, which had limited expressiveness, and some some parts of the models needed to be manually implemented in the end. Anyway, there was a prototype implementation of a workflow engine based on AMFIBIA – which was developed by a group of eight master students over a year – amounting to an effort of about four person years.

We do not go into the details of the models of AMFIBIA here, since the essence of these models is captured in the ECNO models now, some of which are discussed in the next section.

6.2.2 ECNO models of the worklfow engine

In this section, we give an overview and an idea of the ECNO models of Jesper Jepsens master's project [26]. Note that this is an overview only, and that we discuss the most relevant excerpts of these models only. The ECNO Workflow Engine and all its models are deployed together with the ECNO Tool, so that you can have a look at the actual the models and all their details yourself. You will find them in the "model" folder of the project `dk.dtu.imm.se.ecno.workflow.engine`[2]. Note that the models that are shown and discussed here are slightly adjusted and split up into different diagram in order to make it easier to discuss and understand them. In the project, there is a single monolithic coordination diagram only (see discussion in Sect. 6.2.5.2).

6.2.2.1 Core model

We start with discussing the core concepts of business process models and their behaviour in this section. These concepts are independent from the

[2]Remember that you can import this project to the workspace by opening the Eclipse "Plug-Ins" viewer, selecting the project and choosing "Import As" → "Source Project".

different aspects of business process models and independent from the formalisms implementing the modelling notation for the different aspects. This was actually the driving force and the spirit of AMFIBIA [1, 2].

Figure 6.2 shows the class diagram with the core concepts of ECNO. The most important concepts are shown in the two rows at the bottom (shaded in light yellow). The two top rows (shaded in grey), show some more technical infrastructure, which allows us structuring, and accessing business process models and their different aspects, registering them with the engine, and maintaining their runtime information. In our discussion, we focus on the concepts at the bottom. Actually, the diagram is also split vertically: the left-hand side, shows the concepts of the business process models (modelling time); the right-hand side shows the concepts of instances of business processes (runtime). Having meta models for the modelling concepts for BPM as well as for the runtime information on the one-hand side, and clearly separating the runtime information from the model on the other-hand side, is one of the main principles that werw introduced by AMFIBIA already; note that the runtime information refers to the information of the models, but not the other way round. The process models "do not know" which instances of them are running, but instances "know" the modelling concepts they are an instance of.

At the heart of AMFIBIA and also the ECNO workflow engine are the four concepts Process, Task, Case, and Activity, where Case represents one running instance of a Process (indicated by the respective reference process) and Activity represents one running instance of a Task (indicated by the respective reference task). On the modelling side, the main concepts are *processes* and *tasks*: a business *process* model may consist of any number of *tasks*, which, at runtime, are reflected by *cases* and their *activities*.

Note that the core concepts do not yet represent in which order the tasks (or actually the corresponding activities) are supposed to be executed in the respective instances of a process (which we call cases). Neither do the core concepts represents who is allowed to initiate or execute activities, or which data are needed for or are produced by the activities. All this will be represented by models that represent different aspects of a business process. In the ECNO workflow engine, the three main aspects from AMFIBIA are covered: *control*, *organisation*, and *information*. We discuss the concepts for some of these aspects later in Sect. 6.2.2.2.

The core concepts do not cover any concrete aspect yet. But, they provide the infrastructure so that a Process can consists of different parts that represent the concepts concerning the aspects – in the models as well as at runtime. The respective concepts are shown in the the left-most column concerning models, and in the right-most column concerning the runtime information for the running instances: a process model refers to the models for the different aspects; likewise the case and the activity contains the runtime information for the different aspects. Note again, that the runtime

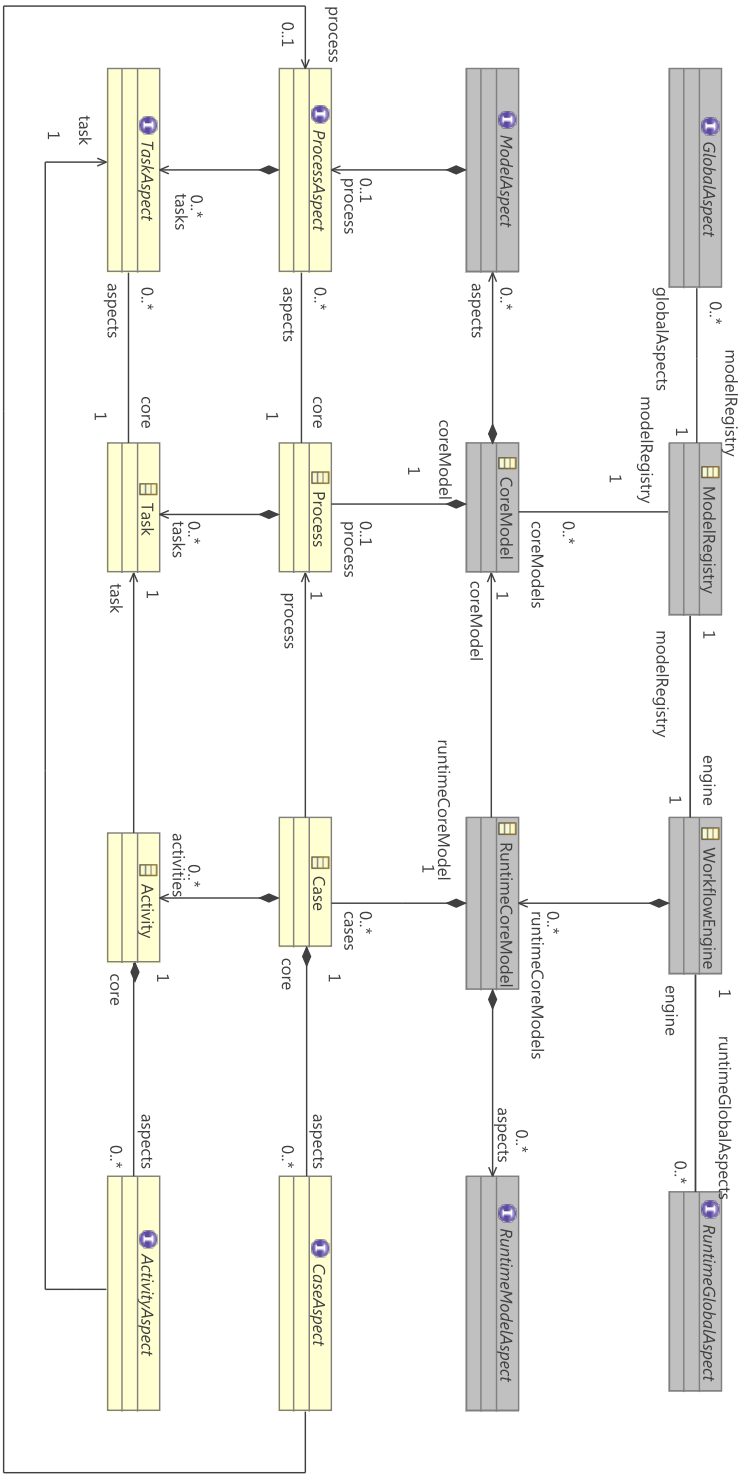

Figure 6.2: BPM: Core concepts

information can refers to the models, but not the other way round.

Note also that all the concepts for aspects are interfaces only. This means that specific concrete versions of them need to be defined when defining an aspect. We come back to this later in Sect. 6.2.2.2.

Now, let us have a brief look at the behaviour concerning the core concepts. Figure 6.3 shows the coordination diagram for the core elements. This diagram defines the event type CreateCase for creating a new instance of a process, which is a Case for the respective process; moreover, the diagram defines the event types StartActivity and FinishActivity, which represent starting and finishing an activity within a process resp. case. Note that there is a slight asymmetry between the event types StartActivity and FinishActivity: StartActivity is triggered from a Case, whereas FinishActivity is triggered from the Activity itself. The reason is that, when starting an activity, the activity does not exist yet; therefore, it needs to be created from somewhere else: the Case. The Activity can, however, take responsibility for its own termination. Note that the event CreateCase is dealt with in a slightly different way – the Case, actually, seems to trigger itself. This is a minor modelling trick: the workflow engine always keeps one fresh instance of a case for each process ready, which is activated when the initial activity of the case is started; the StartActivity will then by synchronized with the CreateCase event, which in turn will initialize this case and create another fresh instance of a case for that process for the next case to be started. Conceptually, this reflects the fact that starting one of the initial activities of a case, actually, starts the case.

The coordination diagram of Fig. 6.3 also shows the coordinations concerning these event types, most of which are straight-forward. The most important part is in the runtime part: the StartActivity and FinishActivity require all the aspects of the respective concept to participate in that event. This way, the coordination makes sure that activities are started and finished only when all aspects are ready for that (for example, an activity can be started only when the control allows to start it, all the data are ready, and there is an agent who is allowed to perform this activity).

Most of the life-cycles of the core concepts are trivial – meaning that all events are possible anytime (there are some minor twists, though which we do not discuss here). The only interesting local behaviour is the one for Case and Activity, which are shown in Fig. 6.4 and Fig. 6.5, respectively.

We start discussing the life-cycle of the Case (see Fig. 6.4). Remember that there is always one fresh case, which is ready for being activated by starting one of its initial activities. This case will be activated by a CreateCase event, which actually is jointly executed together (synchronized) with a StartActivity, which starts the first activity of the case at the same time. After that, the Case can participate in further StartActivity events without synchronizing it with another CreateCase event. In a running case, StartActivity events can happen as long as the case is not finished, which is

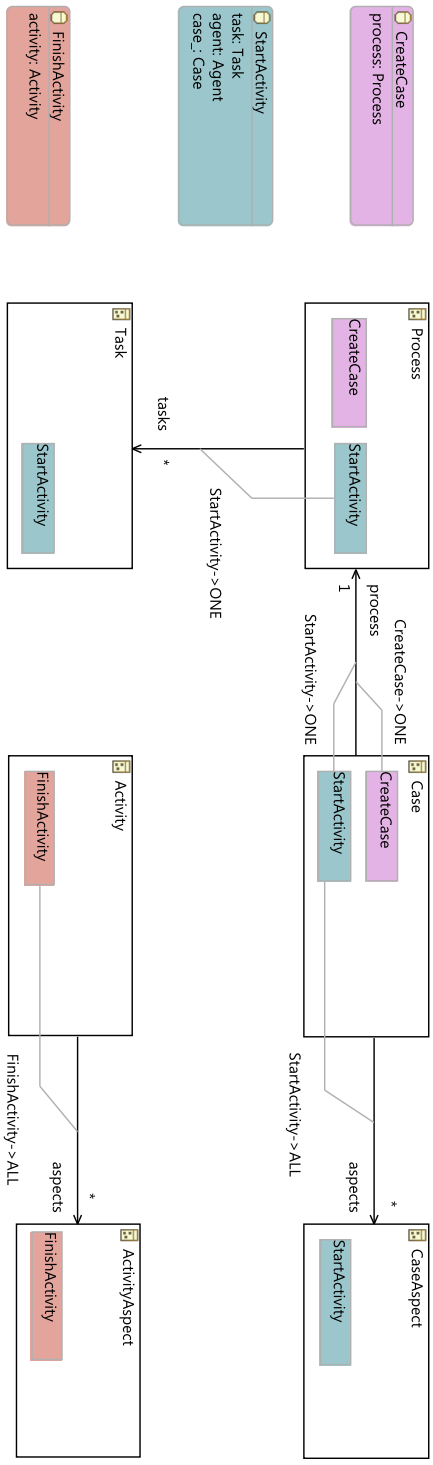

Figure 6.3: BPM: Coordination diagram (core)

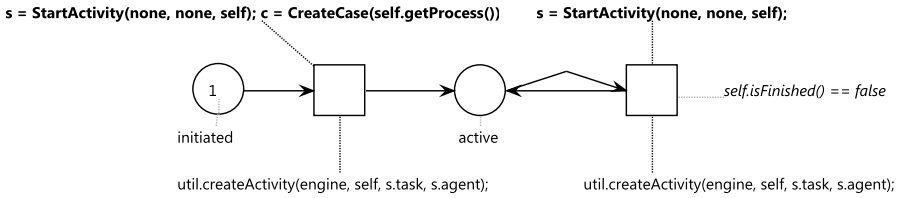

Figure 6.4: BPM: local behaviour of Case

represented by an additional condition. Actually, the termination of a case could have been made an explicit part of the life-cycle with a FinishCase event executed together with a finishing activity. But, the current model does not do that.

The notation used for the ECNO net in Fig. 6.4 and all other ECNO nets of this example might be a bit unfamiliar. The reason is that we show the exact version of the ECNO nets of the master thesis, which uses an outdated notation[3] for assigning parameters to events. This notation exploits the order of the parameters of the event (instead of referring to the parameters by name); the keyword `none` as an expression for a parameter indicates, that no parameter is assigned to the respective parameter.

Figure 6.5 shows the life-cycle of the Activity. This is almost trivial, making sure that every activity can finish only once.

6.2.2.2 Models for aspects

Next, we discuss some of the models concerning the different aspects of business processes. Note that we restrict this discussion to the control aspect,

[3]The current version of the ECNO tool works with both notation, but the outdated one used here might eventually be phased out. With inheritance on events, the order of parameters might change when other packages with events from which this event is inheriting changes; resulting in unforeseen effects in completely unrelated packages. Therefore, the current version of ECNO nets refers to the parameters explicitly by their names. And parameters not assigned do not need to mentioned, which makes ECNO nets more readable.

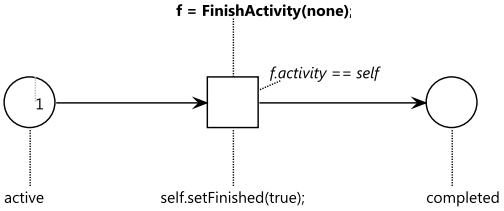

Figure 6.5: BPM: local behaviour of Activity

and a part of the organisation aspect. The ECNO workflow engine covers the information aspect too, but we do not discuss this aspect here.

We start with the discussion of the control aspect as well as one formalism for modelling the control aspect of business processes: Petri nets. Note that AMFIBIA set out to separate the concepts of an aspect from the realization of these concepts in a concrete modelling notation. Anyway, we discuss the control aspect together with the concrete modelling notation here.

Figure 6.6 shows the general concepts of the control aspect as well as how these concepts can be captured by the Petri nets formalism – actually by *workflow nets* [56, 54]. The classes in the top row (in light yellow) represent the concepts from the core model as seen in Fig. 6.2 again. The classes in the two rows below (in light blue) represent the general concepts of the control aspect, and the classes below that (in magenta) show the concepts of Petri nets implementing the concepts of the control aspect. Like before, the classes on the left-hand side represent the modelling concepts, whereas the classes on the right-hand side represent the runtime concepts.

The class TaskC represents the control aspect of a TaskAspect, which in Petri nets is realized as a Transition. On the runtime side, the class ActivityC represents an ActivityAspect from the control point of view, which in turn refers to the control aspect of the case CaseC. The most important part of the control aspect is that a case has a concept of a State, which – as we will see later – determines which activities are possible to be started in the current situtation. In Petri nets, the State is realized as a Marking, which is represented by a set of tokens associated with some places of the Petri net. The model for Petri net roughly resembles the model of Petri nets that we had seen in Chapter 2 in Fig. 2.3 on page 31 – admittedly, it is a bit more ad hoc. The most important difference is that tokens are not contained in places anymore: tokens are contained in a marking, where the tokens refer to the places they belong to. The reason for detaching tokens from places in this model is that tokens represent runtime information, which the actual model should not know about. This leaves the question of how the initial marking of the Petri net is represented in the model itself. To this end, we exploit a speciality of workflow nets: they always start in a specific marking. So, the model does not need to represent the initial marking; we just need to represent the `start` place and the `finish` place of the net by resp. references from the PetriNet[4]. Transitions enabled when a single token is added to the start place correspond to the initial task of a process, which implicitly starts the process (as discussed earlier).

The more interesting part of the models for the control aspect concern the behaviour at runtime. The corresponding coordination diagram is shown in Fig. 6.7. Starting an activity requires the control aspect of the case (CaseC)

[4]In workflow nets, the start place is often called i and the finish place is called o. But, in our meta model of workflow nets, the start and finish places are made explicit.

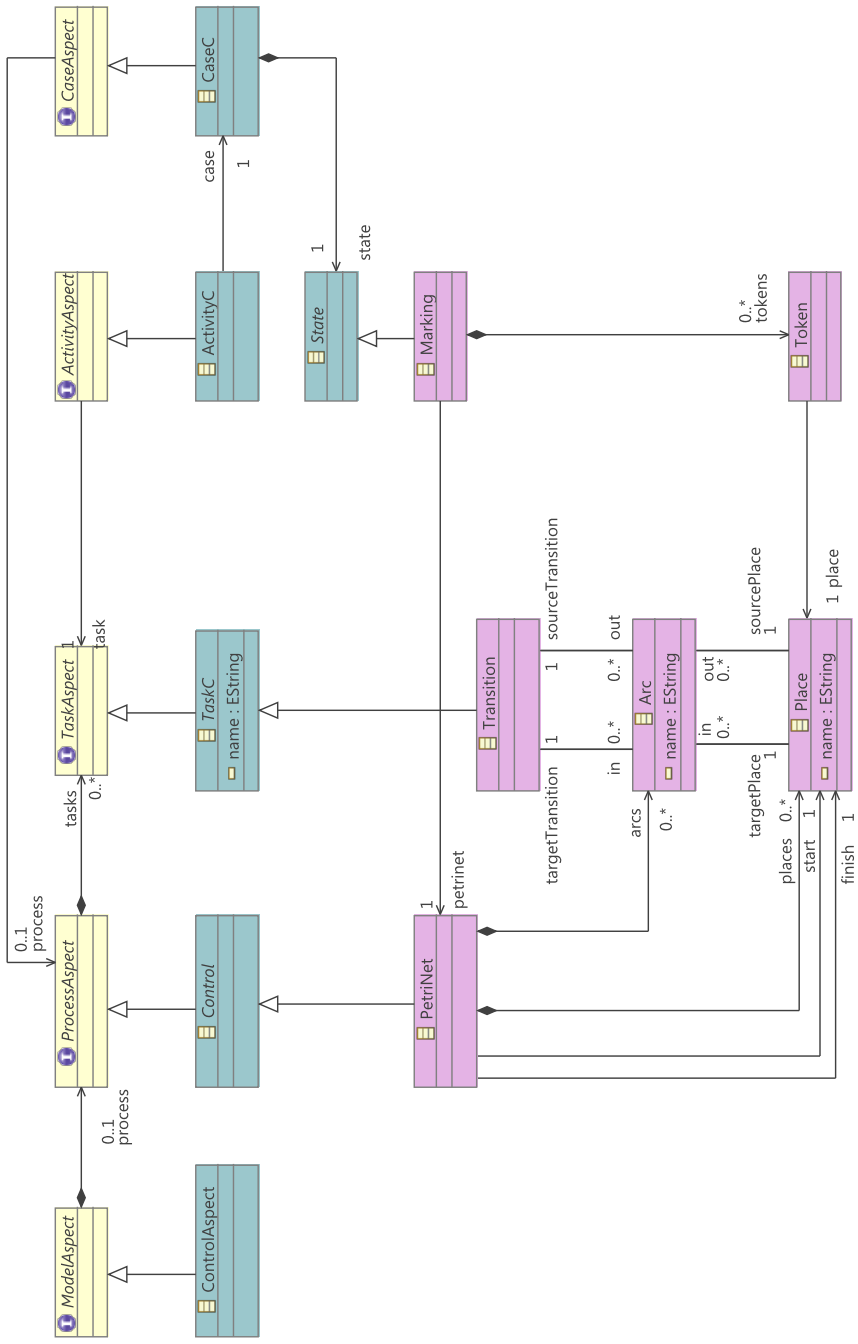

Figure 6.6: BPM: Concepts for control and Petri net formalism

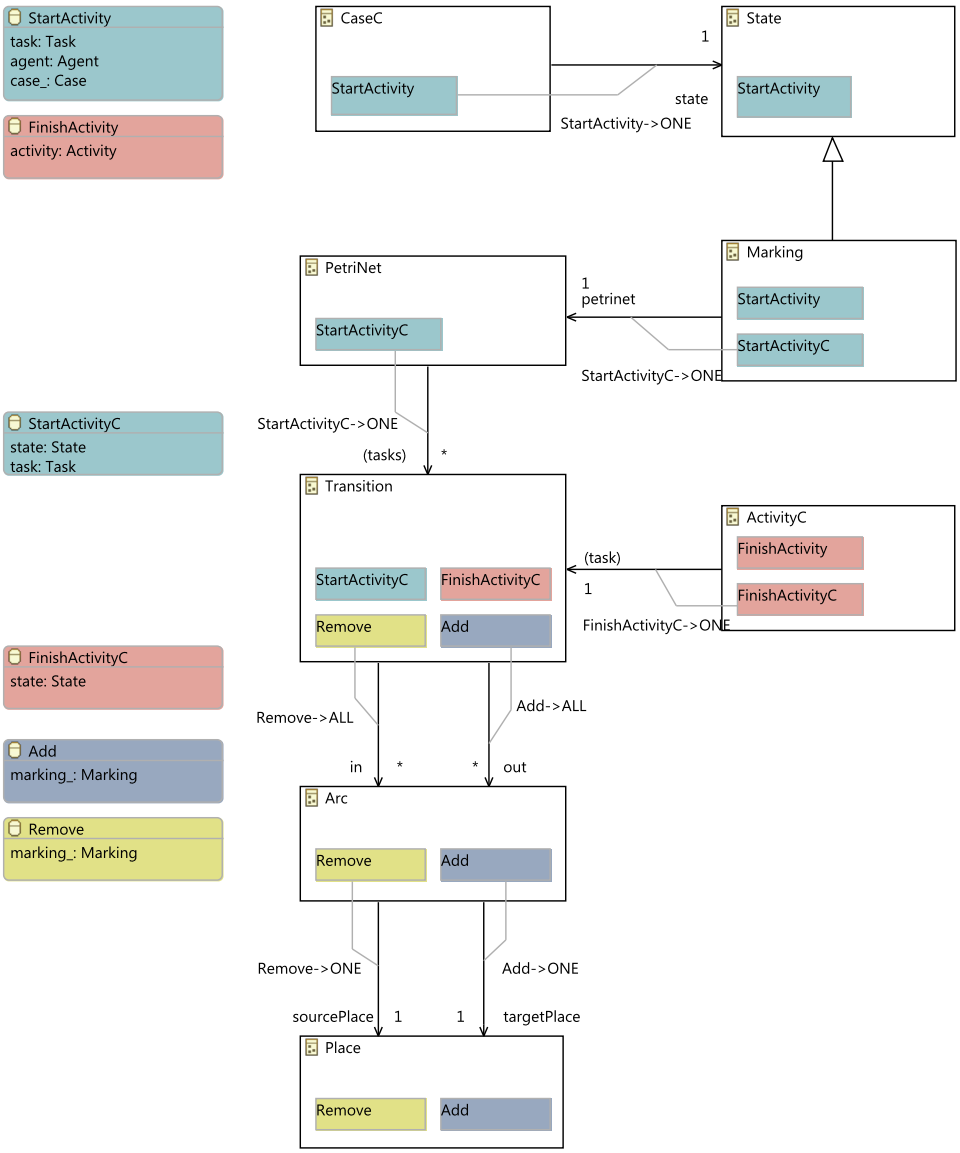

Figure 6.7: BPM: Coordination diagram for the control aspect

to coordinate the event StartActivity with its State – which requires a synchronisation with another event StartActivityC, which represents the control aspect of the event StartActivity[5]. The state in turn requires the model for the control aspect – in our case the Petri net – to participate in the StartActivityC, which requires a Remove event on a Transition to be part of the coordination. The Remove event is used and handled in a similar way to the Petri net example that we had discussed in Sect. 2.1.3.2. Note that in contrast to the original semantics of Petri nets which fires a transition instantaneously, workflow nets are executed in two steps: Starting the activity removes the tokens from the input places, whereas finishing the activity adds the tokens to the output places.

The start of an activity needs to be issued from the case since an instance of the activity is created only upon starting it. By contrast, the activity can take care of its own termination. The ActivityC coordinates a FinishActivityC event with the respective transition, which in turn coordinates it with an Add event which adds all the tokens to the postset of the transition – similar to the Petri net semantics that we had discussed earlier.

The only interesting local behaviour is in the life-cycles of the objects of the Petri net. Since these are similar to the ones discussed in Sect. 2.1.3.2, we do not discuss them here. All the other local behaviours are quite simple. Most importantly they synchronize the StartActivity event with the, StartActivityC, and likewise the FinishActivity event with the, FinishActivityC event.

Next, let us have a brief look at the organisation aspect. Since the structural models are quite straight-forward, we discuss the coordination diagram for this aspect right away. Figure 6.8 shows the coordination diagram for the organisation aspect. Basically, the organisation aspect for the case CaseO delegates the StartActivity event to one of the (possibly) involved Agents; likewise the organisation aspect of an activity ActivityO delegates the FinishActivity event to the Agent to which this activity was assigned.

The ECNO net for the life-cycle of the Agent is shown in Fig. 6.9. From the organisation point of view, an agent can start any activity as long as it does so on its own behalf, and the organisation model allows the agent to do so, which is represented by the additional conditions.

6.2.3 Worklist GUI

The purpose of a workflow engine is to enact the processes, and allow the agents to see the activities they could participate in, and to initiate and finish them. Concerning the information aspect, the agents should also be

[5]At the time when Jesper Jepsen developed the ECNO workflow engine, not all concepts of ECNO had been implemented yet – in particular event extensions were not supported. With event extensions, we would not use two independet event types any more and synchronize them. We would make StartActivityC an extension of StartActivity.

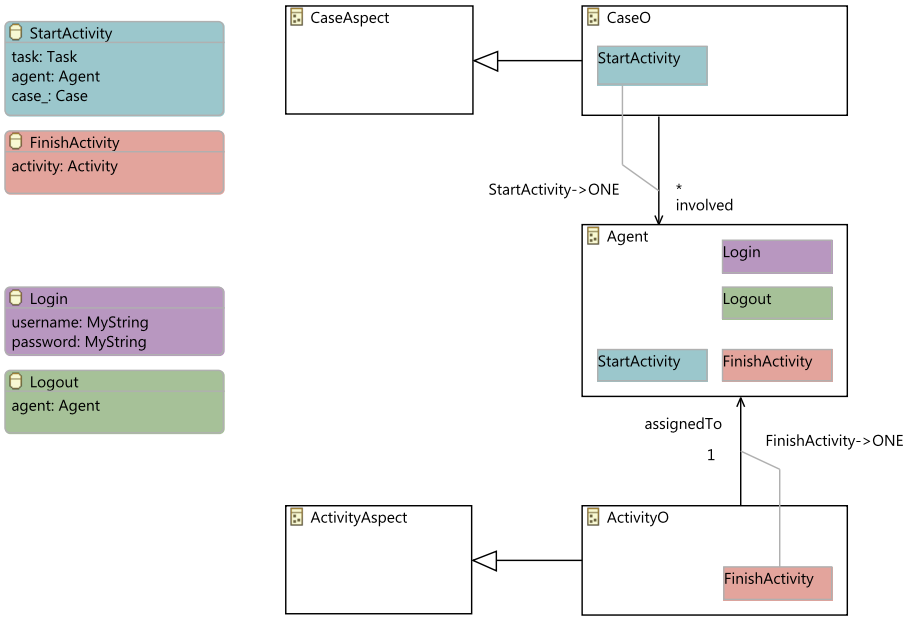

Figure 6.8: BPM: Coordination diagram for organisation

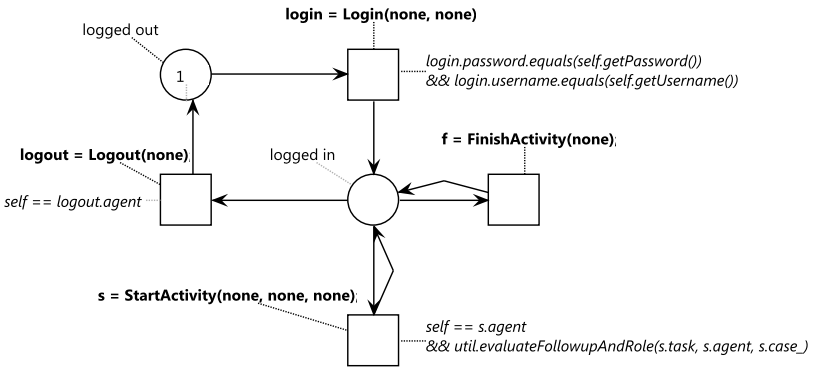

Figure 6.9: BPM: local behaviour of Agent

able to add new documents and change some information of an activity in which they are involved.

In order to allow the agents to do this, the ECNO workflow engine comes with a GUI. This GUI was implemented manually, using the ECNO framework for implementing such GUIs as discussed in Sect. 5.3. We do not discuss this implementation here. The source code can be found in the package `coded.gui` folder of the project `dk.dtu.imm.se.ecno.workflow.engine`.

In the next section, you can find some screenshots of that GUI when we discuss how some example workflows can be enacted with the ECNO workflow engine.

6.2.4 Enacting the example processes

In this section, we give a feeling on how the ECNO workflow engine can be used. This is not so much meant as a user manual – after all, this is a prototype only. But, it is meant to better understand, which functionality is covered by the ECNO models and the workflow engine generated from them.

6.2.4.1 Example process

To this end, we need to have some example processes. Since the focus of Jesper Jepsen's project was on the workflow engine and not on the editors, there are no graphical editors for the different workflow models yet; the models deployed together with the ECNO workflow engine are in abstract syntax only – created by the default EMF tree editors. You will find them in the "example" folder of project `dk.dtu.imm.se.ecno.workflow.example`. For discussing these models here, we use an ad hoc graphical syntax.

This example consists of two processes, but we discuss only one of these processes here: a process for ordering a book from some online book shop. Figure 6.10 shows the workflow net of that process. Initially, a customer can "Place a New Order" for a book, which is then processed ("Take Order"); depending on the outcome of "Take Order", the book shop needs to order the book from a some supplier ("Back Order"), or the book can be "Shipped" right a away; in case of the payment information being invalid, the book shop might "Reject" the order. In the end, the customer can "Check the Receipt". All the time, while the case is not finished yet, the customer can "Track" the status of the placed order.

When discussing the workflow net above, we mentioned already who would be involved in which task – even though the model for the control aspect does not cover that at all. This part is defined in the organisation aspect. We discuss the organisation aspect of that process together with the information aspect, which are shown in Fig. 6.11.

The labels "Customer", "Librarian", "Buyer", "Accountant", "Shipping Agent" attached to the different task, specify who is allowed to do these tasks. These labels represent the *roles* of agents that are supposed to take this task; as part of the runtime configuration, the organisation aspect defines which agent may act in which roles. In our example configuration, Jack is a "Customer", Ellen is a "Librarian", Tom and Tim are "Buyers", and Max is a "Shipping Agent".

Figure 6.11 also shows a part of the information aspect: which documents are input and output for which tasks, as well as some additional pre- and post-conditions for the activities to start and to finish.

6.2.4.2 Running the processes

As said above, these (and some more detailed) models can be found in the "example" folder of project `dk.dtu.imm.se.ecno.workflow.example`. The file "load.behaviourstates" combines all the model and runtime information, along with configuring the ECNO engine and its GUI. This is a configuration file of the ECNO engine as discussed in Sect. 5.4.1, and the ECNO workflow engine can be started from this file as discussed in Sect. 5.4.2 by a right-click on the file and then selecting "ECNO→Start ECNO Engine".

Once you start the ECNO engine on the file "load.behaviourstates", a worklist as shown in Fig. 6.12 pops up, and, in the ECNO Engine registry, you will see that a new ECNO Engine started on that file.

Once you type in a legal user name with a legal password, the log in button will be enabled. Remember that there are users Jack, Ellen, Max, Tim, and Tom with the roles as discussed in Sect. 6.2.4.1. For all these users, the password is "pw".

Initially, we suggest to log in as Jack, since he is the customer who can initiate the book order. In order to log in as a different user later, you can either log out and log in as the new user in a WorkList GUI. But, for ease of use, the Work List GUI allows you to create more instances of Work List GUIs (by pressing the button on the top-left below the headline). This way, you are able to see the work lists for different agents simultaneously.

Figure 6.13 shows the Work List after Jack logged on and selected the "Start a new Online Book Purchase Process". By pressing start, a new case for that process is created.

Now, Jack will have the "Place New Book Order" activity assigned[6] to him in the work list as shown in Fig. 6.14. When selecting it, the "Open" button will become enabled; note that the "Finish button" is not enabled yet, since the information that is required for this task to finish is not yet there.

Pressing the "Open" button will open a window for that Task, which is shown in Fig. 6.15. The tabs represent the different documents associated

[6]Remember, that initiating a case means initiating its initial activity.

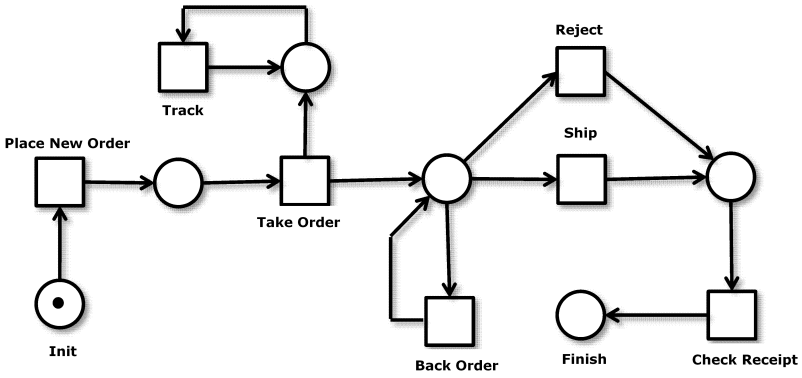

Figure 6.10: Workflow net for control aspect (Source: Fig. 7.1 of [26])

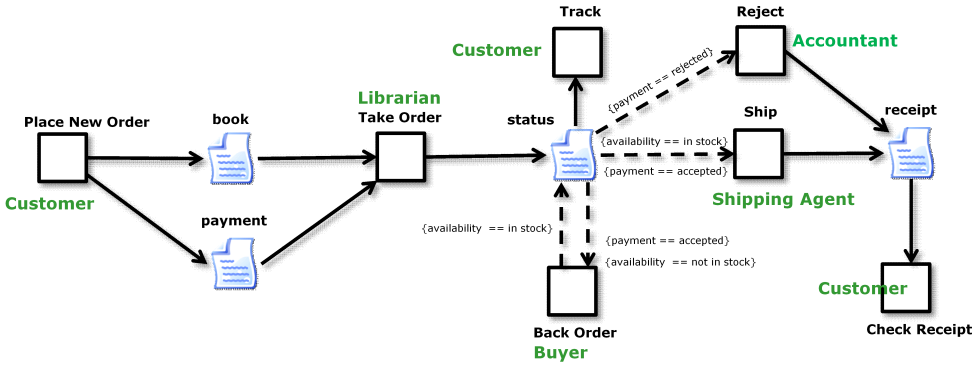

Figure 6.11: Information and organisation aspect (Source: Fig. 7.2 of [26])

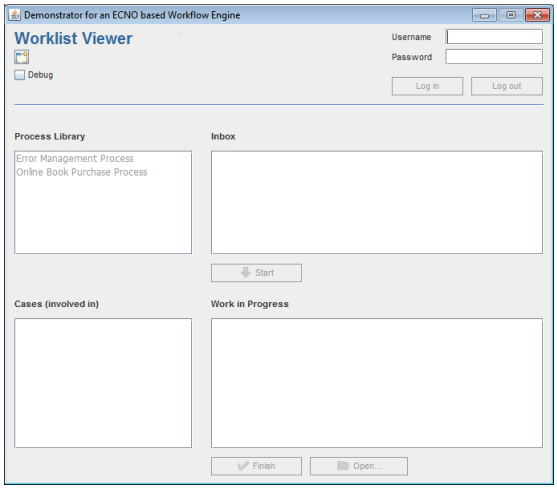

Figure 6.12: Initial Work List View

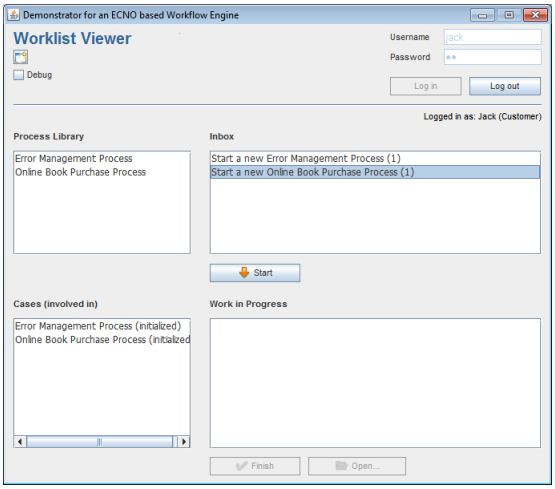

Figure 6.13: Work List View: With Jack logged on

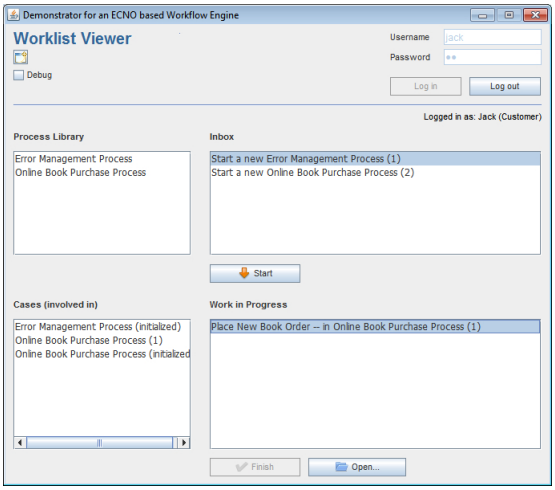

Figure 6.14: Work List View: With activity “Place New Book Order” active

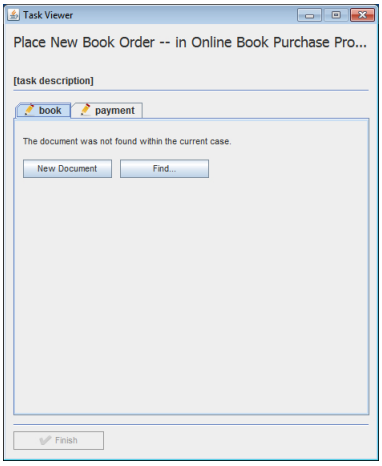

Figure 6.15: Task Viewer for activity "Place New Book Order"

with this activity, which, in this case, both need to be created. When a document can be edited by the agent, a pencil icon is shown. In our example, both documents need to be edited before the task can finish (note that closing the task viewer does not finish the task). In each of the tabs, either a completely new document can be created or an existing one can be selected. For a book, we suggest to select an existing one by pressing the "Find" button. For payment, we suggest to create a new one, and then entering the information to the empty fields of the payment document.

Then you can finish the task by either pressing the "Finish" button in the task viewer or in the Work List.

Next, Ellen could log in to process the order, and later Tom and Max could log on working through the process as discussed in Sect. 6.2.4.1. But, we do not discuss the next steps in detail anymore. You will find all steps to run through a process in the file "README.txt", which comes with the example. It is important, however, that you do not leave the fields in the documents empty – otherwise the case might be stuck.

Note that the work list viewer itself does not provide any means for saving the current state of the workflow engine or for shutting it down. This still needs to be done from the ECNO Engine registry view. In a real workflow engine, there would need to be other views for that purpose. Since the current version of the ECNO workflow engine was meant only to show the feasibility of these ideas, we did not bother to implement such a view yet (see discussion in Sect. 6.2.5).

6.2.5 Discussion

In this section, we have discussed the ECNO models from which a fully functioning workflow engine could be generated. This workflow engine was

developed in a five month master's project by Jesper Jepsen [26]. Though this ECNO workflow engine is a prototype only, it demonstrates that ECNO can be used for larger applications and some more complex behaviour.

6.2.5.1 Prototype not a case study yet

Still, we do not consider this workflow engine as a case study for the use of ECNO in the true sense of the word. There are several reasons for that:

- The workflow engine was developed, when ECNO was still under development itself (it was developed using version 0.3.0), and took benefit of some features of (0.3.2) when this appeared to makes sense. But some features (such as event type extensions) were not used yet. So, ECNO was still a moving target.
- ECNO was developed based on some ideas that came from the AMFIBIA project [1, 2]. The ECNO workflow engine itself, can be considered as a "reimplementation" of AMFIBIA in ECNO. This compromises the universality of this example as a case study.
- The development of the workflow engine by Jesper Jepsen was closely supervised by the originator of the ECNO, which would not be given in a realistic development process.
- In turn, there was not much written material on the ECNO methodology. In fact, this report is the first concise writeup which touches a bit on methodology. But, this is still in its beginning.
- The workflow engine still lacks some features which would be required for its practical use. These missing features and the implications are discussed below in more detail in Sect. 6.2.5.2.
- One very important characteristics of ECNO was not even an issue in the development of the workflow engine: the integration with preexisting parts of the software. The workflow engine was developed from scratch.
- The ECNO tool (and software modelling tools in general) does not support many of the standard functions of modern IDEs, which most programmers are used to and would expect, such as re-factoring, automatic validation, auto-completion, etc.

Therefore, the experience of the project of Jesper Jepsen supports what we believe in: ECNO could significantly speed up the software development process – in particular for software in which events and their coordination play a natural role. A more extensive study providing substantial evidence for such a hypothesis, however, is yet to come.

6.2.5.2 Limitations

As mentioned earlier already, the ECNO workflow engine is a prototype or feasibility study only at the current stage. The current version still has some limitations. We discuss these limitations below.

We start with some limitations, which concern the usability of the ECNO workflow engine only – they do not expose any principle limitations of ECNO.

1. The focus of the current version of the ECNO workflow engine is on the engine part and not on creating and maintaining the business process models (and the different aspects of it). Therefore, there are no graphical editors for workflow nets for the control aspect, or graphical editors for models concerning the organisation or information aspect yet.

   Creating such editors would be supported by different technologies such as the Graphical Modeling Framework (GMF) [14]. Since this is straight-forward (see [29] for some a general discussion) and would not provide much scientific insight, implementing graphical editors for process models were not in the focus of this project.

2. Up to now, the models for the information aspect covers the flow of information within a process only. There is no way to model the information or documents. Right now, some standard document types are used. In a sense, only half of the information aspect can be expressed by models.

   For modelling the other half, standard data modelling techniques, such as for example Ecore models could have been employed. And some of the coordination models would need to be slightly updated in order to deal with information in a generic way. But, since the scientific insight would not be too deep, this half of the information aspect is not yet covered. Moreover, this part of the information model would be related to the missing database support; so, this would best be done together with implementing the database support of ECNO (see discussion later).

Next, we discuss some limitations, which concern ECNO as notation, tool and framework.

1. The work list and the task viewer of the workflow engine were still programmed manually – for the simple reason that ECNO does not provide a notation for defining a GUI yet.

   Developing such a notation for defining a GUI for ECNO applications is planned – but not with high priority, since the scientific insights in developing such a notation would not be too high.

2. The current ECNO workflow engine comes as a single monolithic coordination diagram. The reason for that is that, at the time when the development of the workflow engine started, the ECNO import mechanism was not yet implemented.

   With version 0.3.2 of ECNO, it should be easy to split the ECNO models into different packages now. Actually, splitting up the packages would make the models more elegant, more readable, and more maintainable. But, we did not do this yet.

3. Some parts of the ECNO model still look a bit clumsy. For example, there are two independent events StartActvity and StartActvityC, which basically represent the same event from the core and from the control aspect point of view. The local behaviour takes care of synchronizing these events. This was necessary since the control aspect needed an additional parameter for the StartActivity event which was not provided by the core. With the current version of ECNO (0.3.2), we would rather make StartActvityC an extension of StartActvity – avoiding the need for the local behaviour synchronizing them.

   This could be easily changed and would make the models more elegant; there was no time for doing this yet.

4. The most important limitation of the ECNO framework for practical use is the lack of database support. As of now, the state of an ECNO application, and therefore the ECNO workflow engine, is saved in files (ECNO configuration files and files the configuration is referring to).

   Actually, serialization of the state of an ECNO applications is depending on the underlying object-oriented technology, which is EMF and Ecore in our case. There exist technologies that allow us to load and serialise instances of EMF models to a relational database (e. g. Hibernate combined with Teneo [51]) – generally called *Object Relational Mapping* (ORM). Basically, ECNO would need to be integrated with these technologies, which is a technical issue only – but a non-trivial one. The integration of ECNO with an appropriate ORM-technology is one of the most important items on the road map for the future development of ECNO since this, probably, is the most important feature for the use of ECNO in practice.

In addition, the work on the project showed that devising a notation is not enough. It also needs an explicit methodology, which shows how to work with ECNO and how adequate models should look like. Some quirks of the current ECNO models for the workflow engine, could have been avoided by much better teaching material, a bunch of typical examples, and guidelines for good modelling practices. With this report, we started

making the methodology of ECNO a bit more explicit; but we need many more examples for that purpose.

A last issue, which we did not even touch upon yet, is efficiency. Computing possible interactions can be quite computation extensive. But, in most practical settings, this could be significantly improved due to structural properties of the coordination models. We did not even start to look into this aspect yet. Our first goal with the ECNO workflow engine was to demonstrated the feasibility in principle.

6.2.5.3 Conclusions

Even though the workflow engine cannot be considered as a case study of ECNO yet, it indicates that, by using ECNO and its tool, a piece of software with a significant functionality can be developed in a relatively short period of time.

An empirical evaluation of such a hypothesis is yet to come. Before, starting such a study, the ECNO Tool and the ECNO Framework would need to be equipped with some additional features – most prominently, a persistence mechanism using databases.

If not for implementation, ECNO might serve another purpose: the workflow example shows that it allows us to equip a DSL with a behavioural semantics, which defines the meaning of the DSL in full detail – actually in such a detail that an implementation can be generated from it.

Chapter 7

Conclusion and future plans

In this report, we have discussed the motivation and objective for developing the Event Coordination Notation (ECNO), we have discussed the concepts of ECNO for modelling the local and global behaviour of systems on top of structural models, and we have discussed the ECNO Tool for modelling and generating code, as well as the programming framework behind the ECNO execution engine.

The work on ECNO, however, is far from being complete – conceptually, methodologically, as well as technically. In fact, we are still at the beginning. But, what we have now is a start already. In this chapter, we discuss what we think is achieved already with the current version of ECNO and its tool support and what its important contributions are. More importantly, we discuss some limitations of ECNO and its tool support, and the road ahead in developing and extending ECNO and its methodology.

7.1 What is achieved

One objective of developing ECNO was to be able to model the behaviour of software systems on a high level of abstraction, which is closer to the domain than to the technical concepts for implementing the software. In spite of this high level of abstraction, the objective was that software could be generated from these models fully automatically.

Another more technical objective was the ability to integrating the ECNO models and the software generated from these models with pre-existing software – virtually, with any kind of object-oriented software, manually written or generated by other technologies.

Even though an evaluation of ECNO with respect to these – and some more – objectives is still missing, the different examples discussed in this report, to some extent, support this hypothesis. Actually, closeness to the domain or closeness to the implementation as discussed above would require a carefully designed experiment, which would be a research project in its

own right. But some of the examples seem to formalize the behaviour in an adequate way. Our favourite example is the semantics of Petri nets as discussed in Sect. 2.1.3 and Sect. 6.1. These examples show that the behaviour of Petri nets and signal nets could be concisely captured with ECNO; but also the behaviour of larger systems like a workflow engine could be modelled by using ECNO as shown in Sect. 6.2.2.

We believe that there are three main ingredients to ECNO, which allow ECNO to model behaviour in such a concise way:

- ECNO makes the notion of events explicit as a kind of "object" that exists for an instant only, and the notion of events is clearly distinguished from elements (objects). Still, events or event types enjoy some of the same features as classes: attributes and inheritance. But, events enjoy also some similarities with methods, where attributes can be considered as parameters.

- ECNO models the behaviour of a system on two different levels of granularity: there is the local behaviour for elements which defines the life-cycle of the element; moreover, there is the coordination of behaviour among a set of elements via events, which is defined by coordination annotations. We call this the global behaviour.

  Essentially, the coordination annotations describe who needs to participate in an interaction – it is left to the ECNO engine to compute the valid interactions. This way, interactions are defined in a declarative way.

- The notion of interactions comes with a natural notion of atomicity which defines the things that need to be done together. In turn, interactions that concern a disjoint set of elements, are concurrent. This way, ECNO does not impose a thread oriented way of modelling behaviour. Therefore, ECNO models capture concurrency inherent to some domain in a natural way – without explicitly talking about it.

The main objective of ECNO was to model the behaviour of systems. But, it might also have some nice applications in the area of Domain Specific Languages (DSLs). In DSLs, meta modelling is used to formalize the abstract syntax of a DSL, and constraints are used to formalize the static semantics. With ECNO, it is now possible to formalize the behavioural semantics of such a DSL as well – at least if the DSL has some form of event- or transition-based semantics.

We believe that the major modelling concepts of ECNO for global and local behaviour are in place now. We would expect that we would not need any major changes in the concepts of ECNO – though some adjustments of the concrete syntax and some subtle semantical issues (e. g. concerning inheritance) might be necessary.

7.2 What is missing

Though we believe that ECNO can be used with great benefit as it is already, the tooling and the execution framework still lacks some features for being used in an industrial setting. But, the lacking features and extensions are mostly of a technical nature and do not concern the modelling concepts of ECNO as such. We will discuss some of these limitations below.

In addition to the technical limitations, one thing which is still missing is a methodology which needs to be worked out together with a set of relevant examples and case studies from different application areas.

7.2.1 Integration with databases

As mentioned in Sect. 6.2.5.2 already, the probably most important missing feature for the practical use of ECNO is the possibility of saving the state of an ECNO application in a database and not in a set of files. To this end, ECNO needs to be integrated with some existing technologies for *Object Relational Mapping* (ORM).

Unfortunately, such an integration depends on the underlying object-oriented technology and whether there are ORM-technologies that could be used off the shelf. It would be a start already to integrate ECNO with Hibernate/Teneo [51] for the default technology of EMF. Doing this independently from a specific object-oriented technology would probably be a project in its own right; so we do not plan to do that – at least not in the short run.

7.2.2 DSL for modelling GUIs

As discussed earlier, customized GUIs for ECNO applications need to be programmed manually. For a Model-based Software Engineering approach, which is motivated by getting rid of programming, this is a bit anachronistic – in particular, since GUIs could be generated from models quite early on.

In different student projects, we have already experimented with some simple DSLs that allowed modelling customized GUIs for ECNO applications from which code could be generated. These experiments show, that this is possible in principle (for example, the customized GUI of our workers example that we had discussed in Sect. 5.5.2.1 could be modelled this way). But, the implementation of this DSL is not yet mature enough to be included with the current version of the ECNO Tool. It would probably require a BSc project to develop this DSL and implement a code generator mature enough to be deployed together with the ECNO Tool. A DSL for more advanced GUIs would require even some more effort.

7.2.3 Clearer interface definition

We have seen already that different types of events play different roles in an ECNO application. In our Petri net example, there was the fire even, which directly relates to the Petri net semantics – firing a transition. And there were two other events, add and remove, which took care of propagating the required remove and add actions to the respective places and tokens. These additional events, however, play a subordinate role: they are auxiliary events, helping to realize the fire event. In a sense, they are like auxiliary functions or methods in procedural or object-oriented programming. In particular, to the outside world, i. e. applications using Petri nets, the add and remove should not be visible and it should not be possible to trigger them from the outside directly.

Up to now, ECNO caters for that by declaring some event types as GUI events and some element type as GUI elements. Then the default GUI would only show the GUI elements with buttons for their GUI events. But, this concerns the GUI only. As soon as interactions are triggered programmatically or other ECNO models would have coordination annotations referring to these auxiliary events, these events could be used detached from the fire event. Therefore, ECNO would need a concept of visibility of events outside a package, and eventually ECNO will be equipped with such a feature, which in a way defines an interface to a package. The exact details of such an interface definition and the notion of visibility still need to be worked out – looking at more examples, and taking the ECNO methodology into account.

7.2.4 Performance

The main behaviour of an ECNO application is driven by the element controllers that compute possible interactions, and schedule these interactions for execution (either triggered by the user via the GUI or automatically). The computation of valid interaction, basically, is exploring an AND-OR-tree following all coordination annotations. This can be quite computation intensive and might slow down an ECNO application in which many interactions are going on at the same time.

This problem can probably not be avoided completely. But, sometimes coordinations follow certain patterns, which would allow to pre-compute some possible interactions or would allow to prune the search tree, so that not all possibilities need to be followed up.

Up to now, we did not have a closer look into the possibilities for optimizing the computation of valid interactions. But, it is definitely worth a closer look.

7.2.5 IDE integration

Concerning the modelling and development process, the ECNO Tool is still lacking some features which are expected of today's IDEs.

First and foremost, developers would need a debugger, which would allow to set breakpoints, and to analyse the current state and the currently enabled interaction. Integrating a debugger to the ECNO engine would technically not be a problem. But, developing a debugger would require some conceptual work: what actually is a reasonable break point in ECNO, and how would we characterize the situations at which the debugger should stop? In many cases, we would like the debugger to stop when we are in a situation where we would expect an interaction to be enabled – but it actually is not. How would we characterize such situations? And if an expected interaction is not enabled, how would we visualize why it is not enabled? So, before implementing a debugger, we would need a careful analysis of the typical "bugs" we would like to find by debugging and how to visualize and show them. And there are some indications (see questions above) that this is a bit different from debugging programs in classical programming languages.

Another problem with the current code generator is that it copies some of the code snippets verbatim to the generated code. This applies to all the actions and some of the parameter assignments in the ECNO nets. The editor for ECNO nets does not check these code snippets for syntactical correctness yet. This might result in some syntactical errors in the generated code – which then need to be traced back to the ECNO nets and fixed there. The IDE, should pick up the syntactical errors that are found in the generated code and add error markers to the respective code snippets in the ECNO nets.

Doing this is not very difficult and has been done in some student project for an earlier version of the ECNO Tool. We just did not have the time to align this with version 0.3.2 of the ECNO Tool yet.

Another issue is that some constraints on ECNO models are not yet checked automatically. So, that the violation of these constraints will show only in the generated code.

Generally, we have the feeling that the available IDE support concerning model-based development is at least 10 years behind as compared to available programming environments concerning available functionality (refactoring, code organisation, automatic clean up, ...) as well as quality. Since this applies to all kinds of modelling tools, there is a lot of work and research going on in this area, and we hope that eventually the ECNO Tool can adopt some of the available technology from there – but we do not do specific development or research here, since this would be a completely different kind of research.

7.2.6 Adapters for more technologies

At last, it would be nice, if ECNO would work together with other object-oriented technologies than EMF. This would basically require to implement a package adapter for the respective technology. Also this should not be too difficult, but since there was no concrete need for that yet, we did not have an incentive to doing this for now.

7.3 Road ahead

In this section, we give a brief overview on some of the next steps taken in the development of ECNO – concerning tooling as well as conceptual work. The order of the issues mentioned might give you an idea of the priority of these issues. But, the development will also be driven by needs in concrete projects, available funding, as well as personal interest of volunteers.

7.3.1 Tooling

We start with discussing some issues that concern the ECNO tool support, some of which would of course, also need conceptual work.

- Right now, there is only one kind of exception, which is thrown when an interaction could not successfully be executed: `InvalidStateException`. Actually, there can be quite different reasons for this exception to be throw:
  - First and foremost, the interaction might not be valid anymore at the time when it is executed.
  - Some of the actions which are part of the execution turn out to be impossible and would like to abort and roll back the execution of the interaction.
  - After executing the interaction, a constraint of the model is violated, so that the execution of the interaction needs to be rolled back

  For the controllers that issue the execution of an interaction, it would be useful, if they knew what the actual cause for not executing an interaction was. Therefore, this exception structure will be refined in the future, which will probably require minor adjustments of manually programmed element controllers.
- For the default object-oriented technology of ECNO, which is EMF, we will develop an integration with some ORM technology, so that ECNO applications can store their persistent data in a database instead of files.

Eventually, there might be even some general framework which supports integration with databases for other technologies in a generic way. But, this is a very long-term project, since it is not directly related to the research on ECNO.

- As discussed in Sect. 7.2.3, ECNO will be equipped with a concept of visible elements and event types so that there is a clearer interface, which indicates which parts of the software and model can trigger which kind of interactions, and which elements can coordinate with which other elements with respect to which events.

  In turn, the existing attributes of ECNO identifying an element or an event as GUI element or event will probably be removed (in the first place, these features will be marked as deprecated).

- As discussed in Sect. 7.2.2, we will develop a DSL that will allow us to model a customized GUI for ECNO applications. This DSL should support the most relevant widgets and GUI interactions of modern GUIs; but we expect that some advanced features would still need to be programmed manually.

- The IDE support concerning debugging or properly showing errors in models and many other minor issues will be improved incrementally – resources allowing.

- For larger scale applications, the ECNO models need to be analysed for structural properties, which would allow a more efficient computation of valid interactions. Sometimes, we might even be able to statically compute interactions at compile-time (code-generation-time), so that interactions do not need to be computed at all.

7.3.2 Concepts

Though the tool support is important, we deem some conceptual issues equally important – and some of them even more challenging.

- First of all, we need to work out the ECNO Methodology, which gives clear guidelines when and how to use ECNO and in which way to use it in a beneficial and also maintainable way. This would require many examples, but also identifying some of the typical modelling patterns. And there needs to be material which makes it easy to understand and learn ECNO's modelling philosophy and principles.

- In Chapt. 3, we have presented the semantics of a core fragment of ECNO already. Eventually, we would like to formalize the complete semantics of ECNO for different reasons. First of all, any modelling notation should have a clearly defined formal semantics. Formalizing

the semantics in a clear mathematical way, would ensure conceptual clarity of its concepts.

In addition, we would need a formal semantics for, eventually, developing verification tools, so that ECNO applications could be verified – as certified correct software. But, this definitely is a long-term project.

- Actually, we believe that the semantics of ECNO can be formulated in ECNO itself. This would be philosophically appealing - and make ECNO a meta-modelling notation in the true sense of the word (being about itself), and in a sense show that ECNO is conceptually complete. In turn, modelling ECNO in itself would provide great insights into how to properly use ECNO.

  This is very similar to MOF [44] – only that MOF defines its own syntax only and not its behavioural semantics.

7.4 Getting started

Even though, ECNO still has some limitations , ECNO and its tool support is mature enough to be used for experimentation and for developing software in an academic setting. We hope that this report and the examples deployed with the current version of ECNO will help with getting you started.

ECNO might even be used in an industrial setting, but in this case you would be well-advised being in close cooperation with us. We would be more than happy cooperating with partners doing larger examples in a realistic setting, since this could drive the development of the extensions discussed above, and help working out the ECNO methodology.

Appendix A

Glossary

This glossary provides brief definitions of the most important terms of the Event Coordination Notation, and the underlying technologies. More detailed definitions of these terms can be found by using the Index at the end of this report.

The first part (A.1) discusses the necessary terms from object-orientation, the second part (A.2) discusses the terms and concepts of ECNO itself, whereas the third part (A.3) discusses concepts from the ECNO implementation and its programming framework.

A.1 Terms from object-oriented modelling

In this section, the traditional terms from object-oriented modelling are introduced as far as they are needed for the ECNO notations. For a more detailed explanation of the terms, see Sect. 2.2.1.

Class A *class* in the sense of object-orientation. In *domain models* a *class* typically represents a concept in the application domain, and the *class* distills the common features of a set of *objects*.

Class diagram A model defining the *classes* and the relations between different classes as *references*.

Domain model A *class diagram* or a collection of class diagrams that models the concepts of the application and its domain, rather than modelling the software and its implementation.

Link A *link* is an instance of a *reference*. It represents that an object is related to another object with respect to the specific reference defined in the class diagram.

Object An *object* is an instance of a *class*, which is its type.

Object diagram An *object diagram* represents a collection of *objects* and the *links* between them, which characterizes a specific configuration that meets the requirements defined in the *class diagram* – and sometimes some additional constraints.

Package A *package* is the scope (or namespace) in which classes are defined. In our setting, a class diagram is typically representing a package. A model can comprise many packages, and packages provide a way to structure larger models (conceptually and technically).

Reference A *reference* characterizes a specific relationship between two *classes*. A *reference* is a simple form of UML's associations.

A.2 Terms of the ECNO notation

In this section, the terms and concepts of the ECNO notation and its semantics are defined. For a more detailed explanation of the terms, see Sect. 2.2 and Sect. 4.2.

Action An *action* defines the change made by the *local behaviour* of an element when it participates in an *interaction*. The *action* is a part of the resp. *choice*. The changes might affect the state of the *local behaviour* as well as the attributes and *links* of the underlying *objects*.

Base event Each *event type extension* has a *base event type*, which is the actual event type used in the coordination model.

Behaviour inheritance The local behaviour of an *element* of some type consists of the local behaviours for all the *element types* in the element's type hierarchy. This way an *element type inherits* the behaviour of its super types. Note that the local behaviour can explicitly drop the behaviour of the super types.

Choice In a given *situation*, the *local behaviour* of an *element* defines the enabled *choices* of the element. The choice defines, in which *events* it participates, which values are contributed to the event *parameters*, and the changes made when executed (*action*).

In *ECNO nets*, each choice is characterized by a Petri net transition, together with an *event binding*, a *condition* and an *action*. The choice is enabled, if in the given *situation*, the Petri net transition is enabled and if the condition evaluates to true.

Collective parameter A *collective parameter* of an *event type* is a parameter to which all partners in an interaction can contribute, possibly

different, values. The actual value of that parameter during the interaction is a collection of all the values contributed. By contrast, only one value can be contributed to an *exclusive parameter* of an event.

Condition A *condition* is a boolean expression attached to a transition of an ECNO net, which might refer to attributes of the underlying class as well as to parameters of the events of the event binding. Only if the condition evaluates to true, the *choice* corresponding to the transition is enabled in the given *situation*.

Coordination annotation For a given *element* that participates in an *event*, the *coordination annotations* of the underlying element type defines which other elements need to participate in that event. The *coordination annotations* are attached to references, and either require that *one* or *all* (see *coordination quantifier*) elements of the *links* corresponding to the *reference* need to participate.

Coordination diagram A *coordination diagram* defines all the *element types*, *event types* and *event type extensions* of an *ECNO package*. And it defines the *coordination annotations* for these element types.

Coordination quantifier In a *coordination annotation*, the *coordination quantifier* defines whether *one* or *all* elements at the other end of the respective links need to participate in the respective event.

Coordination set An *element type* can have one or more *coordination sets* for an *event type*. For an *event* of this type, all the *coordination annotations* attached to the *coordination set* need to be met (the *coordination annotations* of other *coordination sets* for that *event type* do not need to be met).

Counting event type An *element type* can declare some of its *event types* (the *event type* the *element type* has *coordination sets* for) as *counting event type*. These are also called *trigger event types*. For these *counting event types*, the element must participate in events of that type as many times as required by some incoming coordination annotations or by the local behaviour. *Event types* of an *element type* that are not *counting*, are sometimes also called *non-counting event types*.

ECNO net *ECNO nets* are a version of Petri nets. An *ECNO net* is one of the ways to to define the *local behaviour* for an *element type* in ECNO. In particular, the transitions of the *ECNO net* define the possible *choices* of the local behaviour in a given *situation*.

Element An *element* is an *object* together with the local behaviour (and its state), which is defined by the *element type* for the *class* of the

object. Note that not all *objects* are *elements*: this is the case, when the ECNO model does not define an *element type* for the *object's class*.

Element type An *element type* is defined on top of a *class*. The *element type* defines the possible coordinations with other *elements* via *coordination sets* and *coordination annotations* as well as the *local behaviour* (e. g. by *ECNO nets*)

Event An *event* is used to synchronize different *choices* and to allow these *choices* to share some *parameters*. Each *event* has some *type* (either an *event type* or an *event type extension*), which define the *event's parameters*.

By contrast to *objects*, *events* exist or live for an instant only during the execution of an interaction and are used for sharing information among different elements participating in an interaction.

Event binding The *event binding* of a *choice* of the *local behaviour* defines, *events* of which *type* must participate in that choice. Moreover, the *event binding* defines which values are contributed to the *parameters* of the different events by the *choice*.

In *ECNO nets*, the *event binding* is attached to the transitions of the *ECNO net*.

Event extension *Event extension* (*event type extension*) is one of the two forms of *event inheritance*. *Event extension* just adds new parameters without actually defining a new *event type* – the actual type of an *event type extension* is called its *base event type*. Different *event extensions* of the same type are compatible (which is not true for the other form of *event inheritance*, which is called *event specialization*).

Event inheritance There are two forms of *inheritance* on event types. The first is *event specialization*, the other is *event extension*.

Event specialization *Event specialization* is one form of *event inheritance*. Two different *specialisations* of the same *event type* are two different and incompatible new *event types*.

Event type An *event type* defines the name and the possible *parameters* and their types. Moreover, the *event type* defines which of the *parameters* are *exclusive* and which of them are *collective*.

Exclusive parameter An *exclusive parameter* of an *event type* is a form of parameter to which only one value may be contributed in an *interaction*. Note that it is possible that different partners of an *interaction* contribute a value to the same *parameter* of the same *event*; in that case, however, the value must be the same.

Global behaviour The *global behaviour* defines how the *local behaviour* of different *elements* is coordinated with each other into *interactions*. In ECNO, the *global behaviour* is defined in a descriptive way by coordination annotations.

GUI element A *GUI element* is an *element* that should be represented in ECNO's GUI. This is indicated by a specific attribute in the *element type*. Note that this attribute might eventually be deprecated in order to separate the actual coordination mechanisms from the presentation.

GUI Event A *GUI event* is an event that should, for a given *GUI element* be visible at the GUI. This is indicated by a specific attribute in the *event type*. Note that this attribute might eventually be deprecated in order to separate the actual coordination mechanisms from the presentation.

Interaction An *interaction* is a set of *events* and *elements*, where each *element* is associated with some *choice*, each of which is associated with some *events*. Some *events* of an *interaction* might be shared among different *elements* of the *interaction*. The *interaction* is valid, if the *choice* of each element is possible according to the *local behaviour*, and if, for each participating *element*, the requirements of its *coordination annotations* are met. A valid *interaction* can be executed, which corresponds to executing the *action* of the *choice* of each participating *element*.

Local behaviour The *local behaviour* of an element defines, which *choices* are possible in a given *situation*. To this end, an ECNO model defines the *local behaviour* for each *element type*. The predominant way of defining the *local behaviour* for *element types* are *ECNO nets*. The *local behaviour* of an *element* is also called the *life cycle* of the *element*.

Life cycle In ECNO, the term *local behaviour* of an *element* and the term *life cycle* of an *element* are used synonymously.

Package An ECNO model can be build up from different *packages*. A *package* can import *element types* and *event types* from other *packages*; cross-references are resolved at run-time.

Parallel (local) behaviour ECNO supports *local behaviour* with some *choices* executed in parallel to each other for the same *element* in a single *interaction*. This is called *parallel behaviour*. *ECNO nets* allow modelling *parallel behaviour* in a natural way (the step semantics of Petri nets): transitions (resp. the *choices* represented by them) that are enabled together, are allowed to fire together. It is up to the specific notation for *local behaviours*, whether they would support *parallel behaviour* or not.

Parameter *Events* can have *parameters*, which are defined in the *event type*.

Priority An *element type* can have different *coordination sets* for the same *event type* or for *event types* that are compatible. In that case, the coordination would require only one of these *coordination sets* to meet the attached *coordination annotations*. Normally, the choice between these *coordination sets* is non-deterministic; but if a *priority* is defined for the *coordination sets*, the *coordination set* with the higher priority takes the precedence. Only if no *interaction* for the *coordination set* with the higher priority can be found, the *coordination set* with the lower *priority* is selected.

Situation A *situation* is a concrete configuration of a system, with an underlying *object diagram* and the *states* for each *element*.

State The *state* of an *element* consists of the state of the underlying *object* (its links and current values of its attributes) together with the state of its *local behaviour*. In ECNO nets, the state of the *local behaviour* is defined by the current marking of the ECNO net.

Top-level event type For an *event type*, its *top-level event type* is the topmost *event type* in its *specialization* hierarchy. Note that this *top-level event type* is uniquely defined, since there is no multiple inheritance wrt. *event type specialization*.

In an *event binding*, it is required that all *event types* have different *top-level event types*.

A.3 Terms of the ECNO programming framework

In this section, the terms and concepts concerning the implementation of the ECNO engine, the ECNO Tool and its programming framework are discussed. For a more detailed explanation of the terms, see Chapt. 5.

Controller configurator A *controller configurator* installs the *element controllers* of the ECNO engine when an ECNO application is started as an Eclipse application.

ECNO application An ECNO engine running an ECNO model (or a set of models) in some configuration is called an *ECNO application*. An *ECNO application* can either be run as stand-alone ECNO Java application or as an ECNO Eclipse application.

Element controller An ECNO application can install *element controllers* with *elements*. These *element controllers*, keep track of which *interactions* are possible for the *element* wrt. to some *event type*. They can

offer these *interactions* at the GUI and execute the respective *interaction* on user request (standard GUI) or they can initiate the execution of an *interaction* autonomously. Typically, an *element controllers* is automatically installed by the ECNO engine by so-called *engine controllers*; but, it is also possible to programmatically install *element controllers*.

Element controllers are used when programming a customized GUI for *ECNO application*; they can also be used for automatically executing interactions when they become enabled.

Engine controller It is possible to install *engine controllers* with a running ECNO engine. These *engine controllers* are notified when new *elements* are encountered by the ECNO engine when computing possible *interactions* or when they are explicitly added to the control of the ECNO engine. Typically, these *engine controllers* are used to automatically install *element controllers* for *elements*. The engine controllers are installed with the ECNO engine on start up of an *ECNO application*. In case the ECNO engine is running as an Eclipse application, the *engine controllers* are installed by a *controller configurator*.

Interaction computation strategy The ECNO Engine computes all possible *interactions* for an *element* and some given *event* (or *event type*). The ECNO engine can be configured with an *interaction computation strategy*. The default strategy is *depth first*; another strategy is *small interactions first*.

Appendix B

ECNO Installation

In this appendix, we discuss how to install the ECNO Tool and the accompanying examples on your computer. This installation guide refers to version 0.3.2 of the ECNO tool with Eclipse Kepler (4.3). Since the ECNO Tool is still developed under Eclipse Indigo (3.7) and was tested with intermediate versions of Eclipse, the ECNO Tool should actually be running on Eclipse Indigo (3.7 and 4.1), Eclipse Juno (3.8 and 4.2), and Eclipse Kepler (3.9 and 4.3).

B.1 Installing Eclipse

The ECNO Tool is based on Eclipse, the Eclipse Modeling Framework (EMF) [8], and the ePNK [31, 35]. Therefore, the ECNO Tool can be installed on any platform which supports Eclipse. Here, we briefly discuss how to install Eclipse version 4.3.

If you do not have installed Java, install Java on your computer first – any version above Java 5 should do. For security reasons, it will probably be best to use the latest version. For more detailed information on how to download and install Java, we refer to `http://www.java.com/`.

Once you have installed Java, you need to download and install Eclipse. We recommend to use Eclipse Kepler 4.3 classic for your platform from `http://www.eclipse.org/downloads`. Extract the downloaded file to the location in your file system to which you want install Eclipse. Then, start Eclipse by starting the executable file called "eclipse" in the folder to which you extracted eclipse[1]. The first time you start Eclipse, it will ask you for a folder where the workspace and all its contents should be saved. In principle, you can chose any location you like; on the Windows platforms, you should choose a very short path (otherwise you might run into problems with too long path names later). Then, click on the arrow icon ("Workbench") on the right-hand side to go to the workspace.

[1] On the Windows platform, the executable file of Eclipse is named "eclipse.exe".

Next, you need to install EMF from the Eclipse update site. In order to do that, select "Help→Install New Software..." in your Eclipse. In the opened dialog, select the Kepler site ("Kepler - http://download.eclipse.org/releases/kepler"). Then, enter the text "EMF" in the filter field. This should reduce the choices of extensions to some EMF features. Select the "EMF Eclipse Modeling Framework SDK" feature and follow through the installation process. Make sure to check the box "Contact all update sites during install to find required software" before your proceed; and remember that you must accept the license at some point in order to proceed.

If you intend to make your own models, we recommend to install a more convenient graphical editor for Ecore models (EMF's light-weight version of class diagrams), which is coming from the *Ecore Tools* project. To this end, install the "Diagram Editor for Ecore (SDK)"[2] feature in the same way as discussed above – "Ecore" as a filter for finding it would do.

B.2 Installing ECNO

When you have successfully installed Eclipse and EMF, you can install the ECNO Tool. Version 0.3.2 is available at the ECNO update site `http://www2.imm.dtu.dk/~ekki/projects/ECNO/download/releases/0.3.2/`.

Select, "Help→Install New Software..." again. In the opened dialog, press the "Add..." button. In the opended "Add Repository" dialog, add a name (e. g. "ECNO Tool") and add the URL `http://www2.imm.dtu.dk/~ekki/projects/ECNO/download/releases/0.3.2/` as location and press "OK". When the dialog closes, this new update site is selected. Select all the available features, and follow through the installation process – remember to "Contact all update sites during install to find required software" before your proceed. Note that the features of ECNO are not signed; therefore, you will be asked whether you would like to proceed.

After the successful installation, you need to restart the Eclipse workbench before you can work with the ECNO Tool – by default, Eclipse will ask you whether you would like to restart it at the end of an update anyway.

B.3 Importing the ECNO Examples

If you had selected all ECNO features when installing the ECNO Tool, you will also have installed all the examples of this technical report. But, they are only installed "behind the scenes". In order to see them in the workspace or explorer of your Eclipse, you need to import them. This can be done with Eclipse's "Import As Source Project" feature.

[2] Note that this feature was called "Ecore Tools SDK", in Eclipse versions prior to Kepler.

To this end, you need to open Eclipse's "Plug-ins" view first: Select "Window→Show View→Other...", and in the opened "Show View" dialog select "Plug-ins" in the "Plug-in Development" category (if you enter "plug-in" to the filter field at the top of the dialog, it will show up right away"). Then press "OK".

After that, you should find the "Plug-ins" view in the view tabs – typically at the bottom of the Eclipse workbench. In this view, you can select the ECNO example or the ECNO examples you want to import. Then, right-click[3] on them and select "Import As→Source Project". After that, the selected example project will be visible in the Eclipse explorer on the left. Note that the imported project is a copy of the example project that is installed behind the scenes; so you can safely change the files in the imported project.

For example, in order to install the introductory example from Sect. 1.3, you would import the project `dk.dtu.imm.se.ecno.example.workers` to your workspace. If you imported this example, your workspace should look like the one shown in Fig. B.1, after the class diagram (Ecore diagram) and the ECNO coordination diagram of the workers example were opened by double-clicking on them. If the editors cannot be opened or the icons

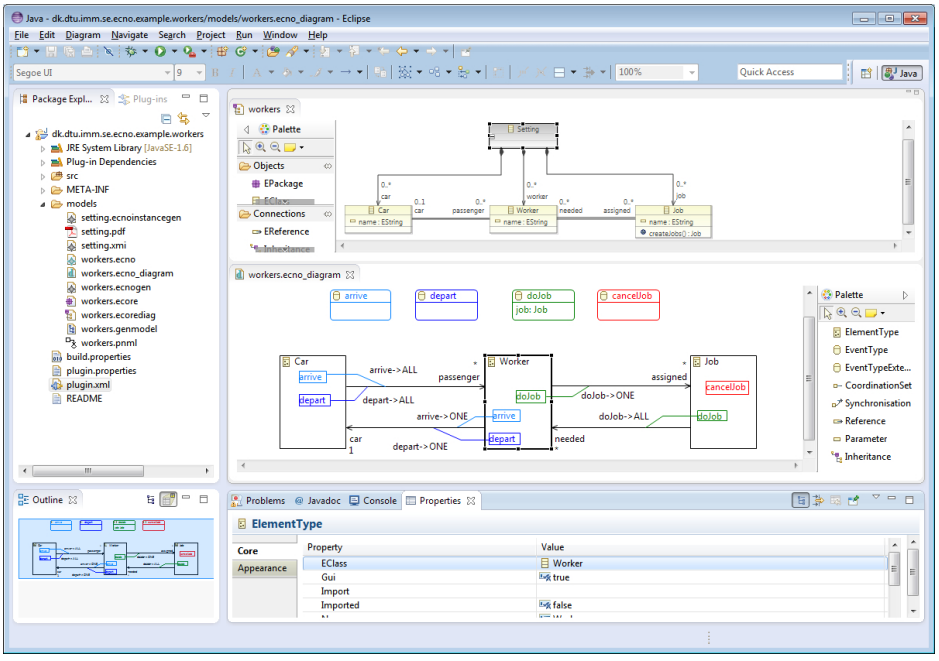

Figure B.1: Eclipse open with the introductory example

for the model files in the folder "models" do not show up properly in your

[3] You can also access the import action via the "File" menu when the respective projects are selected.

Eclipse workspace, your installation probably failed. In that case, try again in a fresh installation of Eclipse.

B.4 Overview of examples

In this section, we give a brief overview of some examples that are deployed together with the ECNO Tool (version 0.3.2) and where they are discussed in this technical report.

`dk.dtu.imm.se.ecno.example.workers` Is the simple workers example, which is discussed in Sect. 1.1. In addition, this example is used in Sect. 1.3 for explaining the basic functions and use of the ECNO Tool.

`dk.dtu.imm.se.ecno.example.workers.worklistgui` This is an extension of the example `dk.dtu.imm.se.ecno.example.workers` above. The ECNO models are slightly extended. More importantly, this example has a customized GUI, which is used for discussing how to implement customized GUIs in Sect. 5.5.2.

`APetriNetEditorIn15Minutes` This is the Petri net example used in Chapter 2. It comprises four different plugin projects:

- `APetriNetEditorIn15Minutes`
- `APetriNetEditorIn15Minutes.diagram`
- `APetriNetEditorIn15Minutes.ecno.gui`
- `APetriNetEditorIn15Minutes.edit`
- `APetriNetEditorIn15Minutes.editor`
- `APetriNetEditorIn15Minutes.runtime`

The most relevant project is `APetriNetEditorIn15Minutes`, which contains the Ecore model and the ECNO models for the Petri net behaviour; project `APetriNetEditorIn15Minutes.runtime` contains some example nets. These two projects are used and discussed in Chapter 2.

The projects `APetriNetEditorIn15Minutes.edit` and `APetriNet EditorIn15Minutes.editor` represent the EMF editor for this example and project `APetriNetEditorIn15Minutes.diagram` represents the GMF editor. These are mostly generated from the EMF model and GMF models. But, the graphical editor is manually extended so that it can be used for graphically animating the token game when the simulation is running. The integration of this graphical editor with the ECNO engine is implemented in `APetriNetEditorIn15Minutes. ecno.gui`, which is used in Chapter 5.5.2.5 for discussing on how to

configure ECNO Eclipse applications. Note that this integration is – as might be indicated by the name "in 15 minutes" – done in a quick and dirty way for demonstration purposes: the command stack for Undo/Redo of the GMF editor does not fully integrate with Undo/Redo of the ECNO execution engine; the simulated net can be edited at runtime; in that case, however, the undo/redo mechanism for undoing interactions does not properly work; using it might eventually compromise the simulator.

`dk.dtu.imm.se.ecno.examples.vendingmachine.split` This is an example of a vending machine, which is used to explain the concepts of inheritance in Chapter 4. Moreover, it is an example which shows how to structure larger models by using packages, and how to extend existing ECNO models. The technical issues of packages are discussed by the help of this example in Chapter 5.1.

This example has three plugin projects that contribute different parts of the ECNO model: `dk.dtu.imm.se.ecno.examples.vendingmachine.split.part1`, `dk.dtu.imm.se.ecno.examples.vendingmachine.split.part2`, and `dk.dtu.imm.se.ecno.examples.vendingmachine.split.part3`. An example instance of a vending machine can be found in project `dk.dtu.imm.se.ecno.examples.vendingmachine.split.part3.runtime`.

Moreover, there are several projects, which are automatically generated from EMF, which concern the EMF editors: `dk.dtu.imm.se.ecno.examples.vendingmachine.split.part1.edit`, `dk.dtu.imm.se.ecno.examples.vendingmachine.split.part1.editor`, `dk.dtu.imm.se.ecno.examples.vendingmachine.split.part2.edit`, and `dk.dtu.imm.se.ecno.examples.vendingmachine.split.part3.edit`. But, you will not need to have a closer look into these projects.

Bibliography

[1] Björn Axenath, Ekkart Kindler, and Vladimir Rubin. An open and formalism independent meta-model for business processes. In E. Kindler and M. Nüttgens, editors, *Workshop on Business Process Reference Models 2005 (BPRM 2005), Satellite event of the third International Conference on Business Process Management*, pages 45–59, September 2005.

[2] Björn Axenath, Ekkart Kindler, and Vladimir Rubin. AMFIBIA: A meta-model for the integration of business process modelling aspects. *International Journal on Business Process Integration and Management*, 2(2):120–131, 2007.

[3] Friedrich L. Bauer and Hans Wössner. *Algorithmic language and program development.* Texts and monographs in computer science. Springer, 1982.

[4] Jan A. Bergstra and Jan Willem Klop. Process algebra for synchronous communication. *Information and Control*, 60(1-3):109–137, 1984.

[5] Gérard Berry and Gérard Boudol. The chemical abstract machine. In *POPL*, pages 81–94, 1990.

[6] Grady Booch. *Object-oriented analysis and design.* Addison-Wesley, 1994.

[7] Johan Brichau and Michael Haupt. Survey of aspect-oriented languages and execution models. Technical Report AOSD-Europe-VUB-01, AOSD-Europe, May 2005.

[8] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, and Timothy J. Grose. *Eclipse Modeling Framework.* The Eclipse Series. Addison-Wesley, 2nd edition, April 2006.

[9] Ruzanna Chitchyan, Awais Rashid, Pete Sawyer, Alessandro Garcia, Mónica Pinto Alarcon, Jethro Bakker, Bedir Tekinerdogan, and Andrew Jackson Siobhán Clarke and. Survey of aspect-oriented analysis

and design approaches. Technical Report AOSD-Europe-ULANC-9, AOSD-Europe, May 2005.

[10] Siobhán Clarke and Elisa Baniassad. *Aspect-oriented analysis and design: The Theme approach*. Addison-Wesley, 2005.

[11] Werner Damm and David Harel. LSC's: Breathing life into message sequence charts. In *Third International Conference on Formal Methods for Open Object-Based Distributed Systems*, FMOODS'99, IFIP TC6/WG6.1, 1999.

[12] Rémi Douence and Jacques Noyé. Towards a concurrent model of event-based aspect-oriented programming. In *European Interactive Workshop on Aspects in Software (EIWAS 2005)*, 2005.

[13] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[14] Richard C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. The Eclipse Series. Addison-Wesley Professional, March 2009.

[15] H.-M. Hanisch and A. Lüder. A signal extension for Petri nets and its use in controller design. In H.-D. Burkhard, L. Czaja, and P. Starke, editors, *Proceedings of the CS&P'98 Workshop*, number 110 in Informatik-Bericht, pages 98–105, Berlin, Germany, September 1998. Humboldt-Universität zu Berlin.

[16] D. Harel and A. Pnueli. On the development of reactive systems. In K.R. Apt, editor, *Logics and Models of Concurrent Systems*, volume 13 of *Series F: Computer and System Science*, pages 477–498. Springer-Verlag, 1985.

[17] David Harel. Statecharts: A visual formalism for computer systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[18] David Harel and Rami Marelly. *Come let's play: Scenario-based programming using LSCs and the Play-engine*. Springer, 2003.

[19] William Harrison and Harold Ossher. Subject-oriented programming (a critique of pure objects). In *OOPSLA*, pages 411–428. ACM, 1993.

[20] L. Hillah, E. Kindler, F. Kordon, L. Petrucci, and N. Treves. A primer on the Petri Net Markup Language and ISO/IEC 15909-2. In K. Jensen, editor, 10^{th} *Workshop on Coloured Petri Nets (CPN 09)*, pages 101–120, October 2009.

[21] C.A.R Hoare. Communicating sequential processes. *Comm. ACM*, 21(8):666–677, 1978.

[22] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[23] ISO/IEC. Systems and software engineering – High-level Petri nets – Part 2: Transfer format, International Standard ISO/IEC 15909-2:2011, February 2011.

[24] ITU-T Recommendation Z.120. Message sequence charts (MSC). ITU, 1996.

[25] Nicholas R. Jennings, Katia P. Sycara, and Michael Wooldridge. A roadmap of agent research and development. *Autonomous Agents and Multi-Agent Systems*, 1(1):7–38, 1998.

[26] Jesper Jepsen. Realizing a workflow engine with the Event Coordination Notation. Master's thesis, Technical University of Denmark, DTU Compute, September 2013. IMM-M.Sc.-2013-101.

[27] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proc. of* ECOOP 2001 – Object-Oriented Programming*,* 15^{th} *European Conference*, pages 327–353. Springer, June 2001.

[28] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.

[29] Ekkart Kindler. Model-based software engineering and process-aware information systems. In K. Jensen and W. van der Aalst, editors, *Transactions on Petri Nets and Other Models of Concurrency II: Special Issue on Concurrency in Process-Aware Information Systems*, volume 5460 of *LNCS*, pages 27–45. Springer-Verlag, 2009.

[30] Ekkart Kindler. Model-based software engineering: The challenges of modelling behaviour. In M. Aksit, E. Kindler, Ella Roubtsova, and Ashley McNeile, editors, *Proceedings of the Second Workshop on Behavioural Modelling - Foundations and Application (BM-FA 2010)*, pages 51–66, June 2010. (Also published in the ACM electronic libraries).

[31] Ekkart Kindler. The ePNK: An extensible Petri net tool for PNML. In *Applications and Theory of Petri Nets -* 32^{nd} *International Conference, Proceedings*, volume 6709 of *LNCS*, pages 318–327. Springer, 2011.

[32] Ekkart Kindler. Integrating behaviour in software models: An event coordination notation – concepts and prototype. In *Third Workshop on Behavioural Modelling - Foundations and Application (BM-2011), Proceedings*, June 2011.

[33] Ekkart Kindler. Modelling local and global behaviour: Petri nets and event coordination. In M. Duvigneau, D. Moldt, and K. Hiraishi, editors, *Petri Nets and Software Engineering. International Workshop PNSE'11, Newcastle upon Tyne, UK, June 2011. Proceedings*, volume 723 of *CEUR Workshop Proceedings*, pages 42–56, June 2011.

[34] Ekkart Kindler. An ECNO semantics for Petri nets. *Petri Net Newsletter*, 81:3–16, October 2012. Cover Picture Story.

[35] Ekkart Kindler. The ePNK: A generic PNML tool - users' and developers' guide for version 1.0.0. Technical Report IMM-Technical Report-2012-14, DTU Informatics, Kgs. Lyngby, Denmark, December 2012. URL `http://www2.imm.dtu.dk/~ekki/projects/ePNK/PDF/ePNK-manual-1.0.0.pdf`.

[36] Ekkart Kindler. The Event Coordination Notation: Execution engine and programming framework. In H. Störrle, G. Botterweck, M. Bourdellès, D. Kolovos, R. Paige, E. Roubtsova, J. Rubin, and J.-P. Tolvanen, editors, *Fourth Workshop on Behavioural Modelling – Foundations and Application (BM-FA 2012), Joint proceedings of co-located events at ECMFA 2012*, pages 143–157, July 2012.

[37] Ekkart Kindler. Modelling local and global behaviour: Petri nets and event coordination. *Transactions on Petri Nets and Other Models of Concurrency*, 6:71–93, 2012.

[38] Ekkart Kindler and David Schmelter. Aspect-oriented modelling from a different angle: Modelling domains with aspects. In 12^{th} *International Workshop on Aspect-Oriented Modeling*, April 2008.

[39] Olaf Kummer, Frank Wienberg, Michael Duvigneau, Jörn Schumacher, Michael Köhler, Daniel Moldt, Heiko Rölke, and Rüdiger Valk. An extensible editor and simulation engine for Petri nets: Renew. In Jordi Cortadella and Wolfgang Reisig, editors, *ICATPN*, volume 3099 of *Lecture Notes in Computer Science*, pages 484–493. Springer, 2004.

[40] Kim Mens, Cristina Videira Lopes, Bedir Tekinerdogan, and Gregor Kiczales. Aspect-oriented programming workshop report. In Jan Bosch and Stuart Mitchell, editors, *Object-Oriented Technology, ECOOP'97 Workshop Reader, ECOOP'97 Workshops, Jyväskylä, Finland, June 9-13, 1997*, volume 1357 of *LNCS*, pages 483–496. Springer, 1998.

[41] Robin Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.

[42] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes (Parts I & II). *Information and Computation*, 100(1):1–40 & 41–77, 1992.

[43] OMG. MDA guide v1.0.1. `http://www.omg.org/cgi-bin/doc?omg/03-06-01`, June 2003.

[44] OMG. Meta Object Facility (MOF) specification, version 1.4.1. Technical Report formal/05-05-05, The Object Management Group, Inc., May 2005.

[45] Chris Peltz. Web services orchestration and choreography. *IEEE Computer*, 36(10):46–52, 2003.

[46] Carl Adam Petri. Kommunikation mit Automaten. Technical Report Schriften des IIM, Nr. 2, Institut für instrumentelle Mathematik, Bonn, 1962.

[47] Wolfgang Reisig. *Petri Nets*, volume 4 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1985.

[48] Wolfgang Reisig. Place/Transition systems. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Central Models and Their Properties*, volume 254 of *LNCS*, pages 117–141. Springer-Verlag, 1987.

[49] David Schmelter. Eine Technik zur Entwicklung und Ausführung aspektorientierter Modelle. Master's thesis, Department of Computer Science, Software Engineering Group, University of Paderborn, Paderborn, Germany, 2007.

[50] P. H. Starke and H.-M. Hanisch. Analysis of signal/event nets. In *Emerging Technologies and Factory Automation (ETFA '97), Proceedings, 6^{th} International Conference on*, pages 253–257. IEEE, September 1997.

[51] Teneo/Hibernate. Eclipsepedia web pages: `http://wiki.eclipse.org/Teneo/Hibernate`, November 2012.

[52] P.S. Thiagarajan. Elementary net systems. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Central Models and Their Properties*, volume 254 of *LNCS*, pages 26–59. Springer-Verlag, 1987.

[53] Rüdiger Valk. Petri nets as token objects: An introduction to elementary object nets. In J. Desel and M. Silva, editors, *Application and Theory of Petri Nets. 19^{th} International Conference, Lisbon, Portugal, Proceedings*, volume 1420 of *LNCS*, pages 1–25, June 1998.

[54] Wil van der Aalst and Kees van Hee. *Workflow Management: Models, Methods, and Systems*. Cooperative Information Systems. The MIT Press, 2002.

[55] Wil M. P. van der Aalst and Twan Basten. Life-cycle inheritance: A Petri-net-based approach. In P. Azéma and G. Balbo, editors, *Application and Theory of Petri Nets*, volume 1248 of *LNCS*, pages 62–81. Springer-Verlag, June 1997.

[56] W.M.P. van der Aalst. Verification of workflow nets. In P. Azéma and G. Balbo, editors, *Application and Theory of Petri Nets*, volume 1248 of *LNCS*, pages 407–426. Springer-Verlag, June 1997.

[57] Michael Wooldridge. Agent-based software engineering. *IEE Proceedings - Software Engineering*, 144(1):26–37, 1997.

Index